Sections atomiques emboîtées avec échappement de processus légers : sémantiques et compilation

Thomas Pinsard

Centre Val de Loire UNIVERSITÉ
Spirit of the Loire Valley

UNIVERSITÉ D'ORLÉANS

UNIVERSITE D'ORLEANS

ÉCOLE DOCTORALE MATHEMATIQUES, INFORMATIQUE, PHYSIQUE THEORIQUE ET INGENIERIE DES SYSTEMES

Laboratoire d'Informatique Fondamentale d'Orléans

THÈSE présentée par :

Thomas PINSARD

soutenue le : **15 Décembre 2014**

pour obtenir le grade de : **Docteur de l'université d'Orléans**

Discipline/ Spécialité : **Informatique**

Sections atomiques emboîtées avec échappement de processus légers : sémantiques et compilation

THÈSE dirigée par :

Frédéric LOULERGUE	Professeur, Université d'Orléans

RAPPORTEURS :

Ludovic HENRIO	Chercheur, CNRS
Fabrice MOURLIN	Maitre de conférence,Université Paris Est Créteil

JURY :

Franck POMMEREAU	Professeur, Université d'Évry
Ludovic HENRIO	Chercheur, CNRS
Fabrice MOURLIN	Maitre de conférence, Université Paris Est Créteil
Frédéric DABROWSKI	Maitre de conférence, Université d'Orléans
Frédéric LOULERGUE	Professeur, Université d'Orléans
Jean-Michel COUVREUR	Professeur, Université d'Orléans

Table des matières

Liste des figures

Introduction

1

Sommaire

1.1 Contexte

1.1.1 Architectures parallèles

De nos jours les architectures parallèles sont partout : de l'ordiphone [1] aux super-calculateurs et fermes d'ordinateurs. Si ce n'est qu'à partir de 2005 que les microprocesseurs multi-cœurs sont apparus pour les équipements de grand public, les architectures parallèles sont développées depuis les années 1960, essentiellement pour les applications de calcul scientifique.

Au milieu des années 1970, les premiers calculateurs *Cray* contenaient des processeurs vectoriels, offrant ainsi du parallélisme au niveau de chaque processeur. Grâce aux instructions vectorielles, en un seul tour d'horloge une instruction pouvait opérer sur plusieurs données, stockées dans un tableau à une dimension. Très rapidement les modèles suivants intégrèrent plusieurs processeurs accédant à une mémoire partagée. Du fait du faible nombre de processeurs, ils pouvaient utiliser la mémoire partagée de manière uniforme.

Avec l'augmentation du nombre de processeurs, un mouvement s'est amorcé qui a conduit à l'utilisation dominante des machines à mémoire *répartie* pour le calcul haute performance. Ainsi en 1993, lors de la première édition de la liste des 500 super-calculateurs les plus puissants du monde, la première place était détenue par la *Connexion Machine* CM-5. C'était une machine massivement parallèle comprenant 1024 processeurs et un réseau d'inter-connection spécifiquement conçus pour ce calculateur. Progressivement, pour des questions de coûts, la conception de processeurs

1. Journal Officiel n°0300 du 27 décembre 2009 page 22537 texte *n*°70

en vue d'équiper uniquement des super-calculateurs a disparu au profit de l'utilisation de micro-processeurs utilisés dans des ordinateurs de grande diffusion. On distingue ainsi actuellement les machines massivement parallèles dans lesquelles l'intégration de dizaine de milliers de processeurs est poussée et repose souvent sur des réseaux d'inter-connection spécifiquement développés pour ces machines, et les grappes dans lesquelles chaque nœud de calcul est très similaire à un serveur conventionnel et le réseau d'inter-connection est plus générique. Il est à noter que le parallélisme n'est pas seulement fourni grâce aux processeurs centraux, les processeurs graphiques (GPU) sont désormais massivement parallèles. Les super-calculateurs les plus puissants actuellement sont d'ailleurs hybrides : ils contiennent à la fois des micro-processeurs généralistes et des accélérateurs adaptés seulement à certains types de calculs mais dont le ratio performance sur coût énergétique est beaucoup plus intéressant.

Longtemps interne aux processeurs et caché au programmeur généraliste, le parallélisme a commencé à être explicite dans les produits grand public, tout d'abord pour des utilisations multimédia : c'est ainsi que les micro-processeurs ont été munis d'unités vectorielles similaires aux premiers super-calculateurs. Un bouleversement plus important, car ayant une influence potentielle sur tous les types d'applications, a eu lieu au milieu des années 2000 avec l'apparition de processeurs multi-cœurs que l'on peut assimiler aux machines parallèles à mémoire partagée. Comme leur prédécesseurs le nombre de cœurs restent limités, pour le moment à une dizaine par processeur. De même les architectures à mémoire répartie ne font pas encore des architectures grand public.

Toutefois on peut constater que si les *architectures* parallèles ne sont plus une niche réservée aux super-calculateurs, la *programmation* parallèle ne se diffuse pas aussi rapidement. Le parallélisme offert aux programmeurs généralistes n'est pas facilement exploité.

1.1.2 Programmation parallèle

On peut distinguer deux grands paradigmes de programmation parallèle : le parallélisme de tâches dans lequel l'accent est mis sur la répartition des calculs et leurs dépendances, et le parallélisme de données dans lequel on s'intéresse en premier lieu à la répartition des données sur lesquels les opérations effectuées sont les mêmes. Il existe de nombreux travaux de recherche et de propositions concrètes de primitives de programmation pour le parallélisme soit sous forme de bibliothèques pour les langages de programmation les plus diffusés, soit sous forme de nouveaux langages.

Dans le cadre du parallélisme de tâches, on peut citer les acteurs et les objets actifs. Un acteur est une entité de calcul qui a un état et qui peut communiquer par envoi et réception de messages (via une « boîte aux lettres »). Le langage Scala [71] en offre une réalisation. Un objet actif s'exécute dans un processus léger dédié. Ce processus exécute uniquement les appels de méthodes sur cet objet actif provenant d'autres objets actifs ou d'objet passif appartenant à cet objet actif. L'appel d'une méthode d'un objet actif est en général asynchrone. L'appel retourne immédiatement un *futur ob-*

jet. L'objet appelant peut ainsi continuer son exécution en parallèle de l'objet actif. Le calcul *Asynchronous Sequential Processes* [14] est une formalisation des objets actifs, et ProActive en fournit une implantation pour Java [7]. Dans le cadre du parallélisme de données, on peut citer les langages data parallèles qui n'ont pas représentant actif à l'heure actuelle.

Le paradigme des squelettes algorithmiques introduit par Cole [21, 73, 76] à la fin des années 80 offre à la fois du parallélisme de tâches et du parallélisme de données. Un squelette algorithmique est une fonction d'ordre supérieur, ou un patron algorithmique, offrant une sémantique séquentielle usuelle mais implanté en parallèle suivant un algorithme parallèle classique, et dans le cas d'un squelette de données agissant sur une structure de données répartie. Les squelette algorithmiques les plus connus sont des squelettes de données : *map* (application d'une fonction à tous les éléments d'une collection) et *reduce* (réduction d'une collection de valeurs par une opération binaire associative) notamment popularisés par leurs variantes MapReduce [30]. Il existe de nombreuses bibliothèques de squelettes algorithmiques [35] : citons simplement Calcium [57] qui est une bibliothèque implantée à l'aide des objets actifs de ProActive, et la bibliothèque de Scala [75] qui peut être vue comme une bibliothèque de squelettes algorithmiques.

Néanmoins, toutes ces propositions sont encore loin d'être répandues. Les deux modèles les plus répandus sont d'une part le modèle de programmation en mémoire partagée avec des processus légers et verrous dont la forme concrète la plus utilisée directement ou indirectement est le standard des *POSIX Threads* ou *PThreads* et d'autre part le modèle de programmation par passage de messages, avec la spécification *Message Passing Interface* [81, 63] qui est de facto un standard pour les machines massivement parallèles et les grappes d'ordinateurs.

Même si ces deux modèles ont été à l'origine inspirés par les architectures matérielles sur lesquelles ils étaient exécutés, on pourrait tout à fait imaginer d'utiliser un modèle par passage de messages pour programmer une architecture à mémoire partagée, et un modèle à processus communicant par une mémoire partagée virtuelle implantée au dessus d'une mémoire répartie.

Cela arrive en pratique pour la première hypothèse : les super-calculateurs actuels sont composés de nœuds ayant une mémoire propre, la mémoire du calculateur est donc répartie, mais les nœuds eux-mêmes sont des machines à mémoire partagée. Il n'est pas rare que ce type d'architecture soit programmé uniquement avec MPI. Le parallélisme sur les nœuds est donc programmé par passage de messages alors que la mémoire est partagée. Les performances sont généralement très bonnes, et c'est souvent plus pour des questions d'optimisation de l'espace mémoire que les concepteurs d'applications passent à une programmation hybride. Dans celle-ci les nœuds sont programmés en utilisant les *PThreads* ou des bibliothèques d'un peu plus haut niveau comme OpenMP [6], et la programmation de l'ensemble des nœuds est faîte en MPI. Notons que c'est vraiment dans ce cadre spécifique de calcul haute performance que le passage de messages est utilisé pour programmer des architectures à mémoire partagée. Pour les programmeurs généralistes, MPI, et le passage de message en général,

est toutefois beaucoup moins connu que la programmation par processus légers et verrous.

Des travaux récents essaient de faire de la seconde hypothèse une réalité, encore une fois dans le cadre du calcul haute performance. Le modèle de programmation *PGAS* (*Partitioned Global Address space*) [5] a pour but d'offrir à la fois un espace d'adressage privé au processus, et un espace d'adressage partagé par tous les processus. Ce modèle essaie de combiner la simplicité du référencement de données du modèle en mémoire partagée et la performance et les localisation des donnée du modèle distribué. De nombreuses propositions de langages conçus pour exploiter un tel modèle mémoire sont apparues au XXI^e siècle : *Unified Parallel C* [83, 69], *Titanium* [66, 85], *Co-array Fortran* [70], *X10* [84, 37], *Chapel* [23, 16]. Un effort de standardisation fournissant une spécification de bibliothèque d'opérations communes permettant d'implanter ces langages a commencé récemment : il s'agit d'OpenSHMEM [17].

Ainsi la programmation en mémoire partagée semble donc toujours d'avenir, même pour les machines à mémoire distribuée.

Au niveau conceptuel, la difficulté de la programmation parallèle en mémoire partagé est l'accès concurrent à des ressources partagées. Il est en général nécessaire d'assurer l'exclusion mutuelle pour qu'un seul processus accède à la fois à une ressource donnée. Les sémaphores ont été un des premiers moyens de synchronisation entre processus concurrents. Ils sont constitués de deux fonctions atomiques permettant de vérifier la disponibilité d'une ressource, et une méthode de signal de disponibilité. Les verrous sont une autre manière d'assurer cette exclusion mutuelle. Une portion de code appelée section critique est protégée par un verrou, qu'il est nécessaire de verrouiller avant d'effectuer toute opération. Une fois la section critique terminée, le verrou est déverrouillée. Un processus voulant verrouiller un verrou déjà verrouillée est mis en attente, ainsi un seul processus peut accéder aux ressources de la section critique si toutes les autres portions de code respectent la convention d'utilisation des verrous. L'objectif de l'utilisation des verrous est ainsi d'assurer l'accès exclusif à une portion de code par un processus donné. Une autre façon de le voir est que cette portion de code doit s'exécuter sans interférence de la part des autres processus.

Les transactions sont un autre mécanisme, pour assurer la non-interférence, et ont d'abord été développées dans les systèmes de bases de données. Le principe général se base sur l'hypothèse optimiste que l'exécution d'une section critique (ou atomique), ne rencontrera aucune interférence. C'est l'hypothèse contraire à celle des système à base de verrous. Si une interférence a lieu, l'exécution appelée transaction, est annulée. Les effets partiels du début de la transaction ne doivent pas être visibles, et la transaction sera ré-exécutée plus tard.

Les systèmes de mémoire transactionnelle [80, 53] sont inspirés des bases de données. Des portions de code sont syntaxiquement marquées comme devant être exécutées sans interférence. Dans ces systèmes on parle plutôt de transactions, et les éléments syntaxiques sont de la forme début et fin de transaction.

Les transaction semblent plus simple à apprendre en comparaison des primitives de synchronisations plus classiques [33]. Cependant, il n'est pas encore clair si

elles peuvent être implantées efficacement [74, 15, 32] et si elles sont réellement plus simples, considérant les sémantiques *formelles* et le raisonnement sur les programmes.

1.2 Contributions

On peut considérer que les marques de début et fin de transaction indiquent simplement une section critique (encore appelée atomique) et que les transactions sont une technique particulière d'implantation permettant d'assurer la non-interférence des sections atomiques.

Pour raisonner formellement, et le plus abstraitement possible, sur les programmes avec sections atomiques il convient donc de s'attacher à la notion de section atomique elle-même plutôt qu'à des réalisations particulières dans des systèmes. Un des avantages souvent cité des sections atomiques est leur modularité. Par exemple il est intéressant d'être capable d'appeler une méthode qui utilise des sections atomiques à partir d'une portion de code qui est elle même dans une section atomique. Dans ce scénario, il n'y a pas nécessairement de parallélisme dans les sections atomiques. Mais la modularité est une nouvelle fois accrue si la méthode appelée est autorisée à créer de nouveaux processus légers. Ce genre de parallélisme interne est interdit ou induit des synchronisations forcées dans les implantations actuelles. Nous affirmons que l'emboîtement de sections et le parallélisme interne sans synchronisations inutiles sont essentiels pour un langage avec sections atomiques.

De plus il est extrêmement important de préciser la notion d'atomicité dans ce contexte. Deux types d'atomicité peuvent être mis en évidence : l'atomicité faible où les sections sont isolés uniquement vis à vis des autres sections uniquement, et l'atomicité forte qui offre une isolation totale. Plusieurs implantations ne sont pas claires sur le modèle qu'elles proposent, et cela peut amener des comportements surprenants. Une façon d'être explicite, est de proposer une sémantique formelle précisant les comportements attendus.

Mes contributions s'articulent autour d'un langage impératif avec sections atomiques emboîtées et avec possible échappement des processus légers créés à l'intérieur d'une section atomique.

Une première contribution est de définir formellement sur des traces d'exécution (qui incluent des évènements d'ouverture/fermeture de section, création/attente de processus, lecture/écriture mémoire), indépendamment de tout langage, la notion d'atomicité et son interaction avec la notion de bonne synchronisation. Je prouve dans ce contexte que toute trace bien formée et bien synchronisé est sérialisable, c'est-à-dire qu'elle est équivalente à une trace où toutes les sections atomiques concurrentes sont exécutées en séquence. Ce résultat a été entièrement formalisé et vérifié avec l'assistant de preuves Coq.

Une seconde contribution est de définir un langage de programmation impératif permettant l'emboîtement de sections atomiques, le parallélisme interne aux sections atomiques, et l'échappement de processus légers. Sa sémantique opérationnelle est

montrée produisant des traces qui respectent les conditions de bonne formation définies dans la première contribution. Ce résultat a été formalisé et partiellement vérifié avec l'assistant de preuves Coq.

Enfin une dernière contribution est la définition d'une passe de compilation de ce langage vers un langage impératif parallèle avec processus et légers et verrous. Je pose également les bases qui permettront de prouver formelle la correction de cette passe de compilation. Cette contribution a été partiellement formalisée en Coq.

1.3 Plan du mémoire

Une partie de mes contributions consistant en la modélisation et la preuve de propriétés à l'aide de l'assistant de preuve Coq, une introduction courte à Coq est fournie dans le chapitre 2. Ce chapitre contient également des rappels de notions de préservation sémantique puisque l'on s'intéresse à la correction de passes de compilation.

Dans le chapitre 3, j'examine les travaux existants en lien avec la notion d'atomicité : les implantations les plus courantes (les transactions) et moins courantes (l'inférence de verrous) ainsi que les sémantiques de langages de programmation contenant des constructions de sections atomiques. Je fais également un tour rapide des travaux en vérification de compilateurs.

Dans le chapitre 4 je me concentre sur la sémantique des sections atomiques. Je considère un langage impératif simple avec du parallélisme *fork/join* et des sections atomiques délimitées syntaxiquement. Elles supportent l'emboîtement et le parallélisme interne où les processus légers peuvent s'échapper de leur section englobante. Il n'y a pas de synchronisation entre la fin d'une section et la terminaison des processus légers démarrés dans cette section. La sémantique du langage est aussi permissive que possible et n'est pas liée à une implantation particulière. Plus précisément, nous considérons des traces de programmes satisfaisant des conditions basiques et, plus important satisfaisant la propriété d'atomicité faible, c'est-à-dire qu'il n'y a pas d'interférence entre sections concurrentes. Je donne une définition de l'atomicité et de la bonne synchronisation et la preuve que cette dernière assure l'atomicité forte (à une équivalence de traces près). Ces résultats sont disponibles en Coq.

Les conditions de bonnes formations représentent des traces d'exécution de programmes qui ont un sens, pour une définition raisonnable d'exécution de programmes. Ces conditions sont une spécification pour les définitions formelles d'exécution de programmes, c'est-à-dire les sémantiques opérationnelles formelles. Je présente dans le chapitre 5, un langage, nommé *Atomic Fork Join* (AFJ) à travers sa syntaxe et sa sémantique opérationnelle. J'établis ensuite la preuve que celle-ci respecte les conditions précédentes, dans le sens ou elle ne génère que des traces bien formées. Une partie de ces résultats est modélisée en Coq.

Dans le chapitre 6 je présente le langage *Lock Unlock Fork Join* ou LUFJ. Ce langage est similaire au précédent pour ce qui est du fragment séquentiel impératif et la gestion des processus légers mais diffère en ce qui concerne les primitives de synchronisation :

il s'agit ici de verrous. LUFJ correspond ainsi à une version simplifiée d'un langage impératif tel que C avec une bibliothèque de *PThreads*.

Le chapitre 7 propose une compilation de AFJ vers LUFJ, c'est-à-dire essentiellement le passage d'une synchronisation par sections atomiques à une synchronisation par verrous. La vérification de cette passe de compilation nécessite de mettre en place des outils pour le raisonnement sur l'équivalence des états des exécutions des deux langages.

Il faut tout d'abord apporter une attention particulière à l'équivalence mémoire. Dans le chapitre 8, pour simplifier nous considérons tout d'abord une transformation source-à-source d'un sous-langage impératif sans parallélisme. Nous mettons en place les outils pour raisonner sur la correction d'une phase de compilation et vérification la correction de la transformation.

Dans le chapitre 9 nous étudions la correction de la passe de compilation définie au chapitre 7. La structure du raisonnement permettant d'établir la correction est explicitée, néanmoins tous les résultats nécessaires n'ont pas encore été prouvés.

Enfin le chapitre 10 établit un bilan et dégage des perspectives de recherche.

L'annexe A détaille le code de macro-commandes et les preuves de certains résultats utilisés dans le texte principal.

Des versions préliminaires d'une partie des travaux présentés dans ce mémoire ont été publiée les actes de colloques internationaux :

— F. Dabrowski, F. Loulergue, and T. Pinsard. Nested Atomic Sections with Thread Escape: An Operational Semantics. In *International Conference on Parallel and Distributed Computing, Applications and Technologies (PDCAT)*. IEEE, 2013.

— F. Dabrowski, F. Loulergue, and T. Pinsard. Nested Atomic Sections with Thread Escape: A Formal Definition. In *ACM Symposium on Applied Computing (SAC)*. ACM, 2014.

— F. Dabrowski, F. Loulergue, and T. Pinsard. Nested Atomic Sections with Thread Escape: Compilation to Threads and Locks. In *ACM Symposium on Applied Computing (SAC)*. ACM, 2015.

Préliminaires

2

Sommaire

Ce chapitre a pour objectif de présenter quelques pré-requis pour la suite de la lecture. Nous commençons par une introduction à l'assistant de preuve Coq, puis nous présentons les techniques de simulation permettant d'établir des propriétés de préservation sémantique, au cours de phases de compilation.

2.1 Une introduction à Coq

Nous allons développer des langages et des sémantiques accompagnés de propriétés prouvées. La manière classique d'écrire une preuve est d'utiliser le langage naturel et des notations mathématiques. Les preuves sont donc dépendantes du style de l'auteur, et des ellipses peuvent être laissées au lecteur. À lui de reconstituer ces parties manquantes pour se convaincre de la correction. Cependant dans certains cas, ces ellipses peuvent masquer en réalité des trous dans la preuve. Pour éviter ce genre de problème, les assistants de preuve proposent une manière d'écrire des preuves plus formelles.

Coq [82, 9, 18] est l'un des assistants de preuve les plus utilisés. Il est basé sur le calcul des constructions (co-)inductives, une extension de calcul des constructions [22]. C'est un outil qui permet notamment de vérifier des preuves, basé sur la correspondance de Curry-Howard [48] qui relie des (arbres de typage de) termes d'un λ-calcul typé avec des arbres de preuves de la déduction naturelle.

La figure 2.1 présente (à gauche) le système d'inférence pour la logique propositionnelle avec pour seul connecteur l'implication, et le système d'inférence (à droite) pour le typage des termes du λ-calcul simplement typé. Cette figure met en relief

Déduction naturelle	λ-calcul simplement typé
$\dfrac{A \in \Gamma}{\Gamma \vdash A}$	$\dfrac{x : A \in \Gamma}{\Gamma \vdash x : A}$
$\dfrac{\Gamma, A \vdash B}{\Gamma \vdash A \to B}$	$\dfrac{\Gamma, x : A \vdash e : B}{\Gamma \vdash (\lambda x : A.e) : A \to B}$
$\dfrac{\Gamma \vdash A \to B \quad \Gamma \vdash A}{\Gamma \vdash B}$	$\dfrac{\Gamma \vdash e : A \to B \quad \Gamma \vdash e' : A}{\Gamma \vdash (e\ e') : B}$

FIGURE 2.1 – *Déduction naturelle et λ-calcul typé*

la similitude de ces deux systèmes. En fait on peut prouver que pour toutes les formules intuitionnistes il existe une preuve de cette formule en déduction naturelle si et seulement si il existe un λ-terme qui a cette formule comme type. La relation entre un programme et son type est la même qu'entre une preuve et son énoncé. Vérifier une preuve est similaire à l'action de typer, un ordinateur peut donc vérifier qu'une preuve est bien la preuve d'une proposition. En revanche trouver une preuve pour une logique un minimum expressive est indécidable : l'intervention humaine est donc nécessaire. Les assistants de preuve, que l'on appelle également prouveurs interactifs, apportent également une aide à l'écriture des preuves.

Dans le cas de Coq, c'est une aide à la construction d'un terme de preuve, qui est un terme du calcul des constructions inductives. Une fois le terme construit, le système vérifie qu'il est bien typé par l'énoncé du théorème considéré. Coq fait parti des outils d'aide à la preuve de théorème comme *Isabelle/HOL, PVS* mais ayant comme caractéristique importante d'obtenir le terme de preuve et la possibilité de générer des programmes certifiés à partir des preuves.

Coq propose un langage de spécification, appelé *Gallina*, pour écrire les types et les programmes. Les fonctions en Coq sont considérées du point de vue algorithmique, c'est à dire qu'elles effectuent un calcul. Un calcul en Coq est une suite de réductions de termes (une expression bien formée) dans une forme irréductible. Une propriété fondamentale de Coq est qu'un calcul doit toujours terminer, une propriété appelée *normalisation forte*.

Coq n'est pas seulement un outil pour vérifier les preuves, mais un assistant de preuve, qui va nous aider à construire une preuve au fur et à mesure, à l'aide de *tactiques*. Ces tactiques sont des commandes que l'on applique pour décomposer ou pour résoudre ce que l'on veut prouver.

Nous allons voir dans un premier temps comment définir de nouvelles structures de données qui nous permettent d'exprimer nos propositions. Ensuite nous examinons la partie logique en détaillant comment définir ces propositions et comment les

prouver, à travers des exemples qui mettent en évidence les tactiques de base. Enfin nous terminons par la présentation d'un moyen de structurer son code.

2.1.1 Programmation fonctionnelle en Coq

Coq permet à l'utilisateur de définir de nouvelles structures de données. Celles-ci sont définies par induction. Les listes par exemple sont définies de la façon suivante :

```
Inductive list (A:Type):Type:=
  | nil: list A
  | cons: A →list A →list A.
```

Ce type possède deux *constructeurs*, nil pour la liste vide, et cons qui construit une nouvelle liste à partir d'un élément et d'une liste. La liste est polymorphe car elle prend un argument A supplémentaire de type Type. Coq est donc un λ-calcul de plus haut niveau car il permet de faire dépendre les termes de types.

Pour utiliser une telle liste, nous devons spécifier le type des éléments. Dans l'exemple suivant nous construisons deux listes de naturels. La première, la liste vide, est utilisée pour bâtir la seconde :

```
Definition empty := nil nat.
Definition l:= cons nat 0 empty
```

Néanmoins il est possible de déclarer le type A comme étant *implicite*, en le plaçant entre { } dans la définition. Coq tentera de l'inférer à partir du contexte.

Il est possible de définir des notations pour faciliter la lecture et l'écriture des définitions et des preuves. Par exemple nous pouvons définir les notations suivantes pour les listes : la notation [] est utilisée pour représenter les listes vides, [x] pour les listes constituées d'un seul élément, la notation infixe :: pour le second constructeur, et la notation [x ; .. ; y] pour la construction de liste par énumération. Nous utilisons ensuite cette dernière notation sur deux exemples de listes, en laissant Coq inférer le type. Il est à noter que le texte entre *(∗∗)* correspond aux commentaires :

```
Notation " [] " := nil.
Notation " [ x ] " := (cons x nil).
Notation " x :: l " := (cons x l).
Notation " [ x ; .. ; y ] " := (cons x .. (cons y nil) ..).
(∗utilisation∗)
Definition l' := [1;2;3;4].
```

Pour opérer sur ces structures de données, il faut définir des fonctions. Ces fonctions utilisent le filtrage de motif comme cela est usuel en programmation fonctionnelle. Par exemple, la fonction tail qui retourne la fin de la liste, va inspecter la liste en fonctionnant par cas sur le constructeur de liste utilisé :

```
Definition tl (l:list A) :=
  match l with
   | [] ⇒[]
   | a :: m ⇒m
  end.
```

Il est nécessaire de spécifier tous les cas possibles (constructeurs) de liste, car les fonctions en Coq doivent être totales.

Nous pouvons également définir des fonctions récursives, en utilisant Fixpoint. La fonction suivante retourne le n^{ieme} élément d'une liste, s'il existe, une valeur par défaut donnée en paramètre sinon :

```
Fixpoint nth (l:list A)(n:nat)(default:A) {struct l} : A :=
  match n, l with
   | O, x :: l' ⇒x
   | O, other ⇒default
   | S m, [] ⇒default
   | S m, x :: t ⇒nth t m default
  end.
```

Là encore, le filtrage de motif est de nouveau utilisé, mais cette fois sur l'indice et sur la liste. La notation utilisée pour les indices est celle des entiers de Peano, S m signifie $m + 1$. Seules des fonctions qui terminent peuvent être définies en Coq. Il faut donc un critère pour assurer cette terminaison. Ainsi dans une fonction récursive, chaque appel récursive doit s'effectuer sur un sous-terme strict (la vérification est syntaxe) d'un des arguments de la fonction. Ici le struct l indique que la récursion se fait sur la liste. Ici l'appel récursif (au quatrième motif) se fait sur t qui est bien un sous-terme strict de l. Notez que l'indice décroît lui aussi à chaque appel, la récursion aurait pu se faire sur ce terme également.

Intuitivement la fonction nth n'est définie que pour les positions à l'intérieur de la liste. Néanmoins Coq n'accepte que les fonctions totales, il est donc nécessaire de la rendre totale, ce qui est fait ici en ajoutant une valeur par défaut qui est retournée lorsque la position n'est pas dans la liste.

Cette définition peut poser problème lorsqu'il n'est pas possible de trouver une valeur par défaut, mais heureusement ce n'est pas la seule façon de rendre totale cette fonction. Il existe en Coq (comme dans d'autres langages fonctionnels) le type option, qui est utile pour décrire une fonction partielle. Pour une fonction partielle dont les résultats sont de type A, nous la définissons en tant que fonction totale, mais dont les résultats sont de type option A.

Ce type est défini dans une bibliothèque standard de Coq de la façon suivante :

```
Inductive option (A:Type) : Type :=
  | Some : A →option A
  | None : option A.
```

Ainsi lorsque la fonction n'est pas définie, elle retourne None, autrement Some

suivi de la valeur. À l'usage, il est nécessaire d'ajouter une étape d'inspection de la forme du résultat lorsque une fonction renvoie une valeur de ce type.

Le nouveau nth est donc défini de la manière suivante :

```
Fixpoint nth_error (l:list A) (n:nat) : option A :=
  match n, l with
    | O, x :: _ ⇒Some x
    | S n, _ :: l ⇒nth_error l n
    | _, _ ⇒None
  end.
```

Nous pouvons voir que lorsque l'indice vaut 0 et que la tête de liste existe, la fonction renvoie cette valeur adaptée pour le type option. Nous pouvons également noter l'utilisation du caractère _ pour désigner la fin de la liste. Celle-ci ne nous intéresse pas, nous n'avons pas à la nommer explicitement, le caractère _ sert donc de joker. Le dernier cas regroupe tous ceux où la tête de la liste ne pourra être retournée, c'est donc None qui est renvoyée.

2.1.2 Preuves

Nous avons vu que Coq offre les possibilités de définition de structures de données et de fonctions comme tout langage de programmation. Examinons maintenant comment Coq nous permet d'écrire des preuves.

Tout d'abord il est nécessaire de déclarer notre proposition. Pour cela nous utilisons les mots-clés Theorem, Lemma ou Fact suivant l'importance de l'énoncé (Coq les considère comme des synonymes). Nous faisons suivre du nom que nous donnons à cette proposition, et enfin l'énoncé. Considérons la formule suivante exprimée de manière classique :

$$\forall P, Q, R, \quad (P \to Q) \to (Q \to R) \to (P \to R)$$

En Coq elle est traduite ainsi :

```
Theorem imp_trans: ∀(P Q R : Prop), (P →Q) →(Q →R) →P →R.
```

Le nom du théorème est suivi de l'énoncé qui est constitué des propositions quantifiées utilisées, et de la propriété. Coq nous demande ensuite de prouver cette dernière.

Quand nous spécifions une proposition, nous n'écrivons généralement pas directement le terme de la preuve en tant que λ-terme. Il est plus facile de raisonner de la même façon qu'on le ferait en utilisant les règles de la déduction naturelle. Cela s'effectue en passant en mode de résolution de preuve interactive de Coq et en utilisant le langage des tactiques Ltac.

La commande Proof (qui est optionnelle) permet de marquer le début de la preuve. Le but initial à prouver est affiché ainsi que le contexte regroupant les hypothèses à notre disposition.

Après avoir entré la tactique intros, Coq indique que son état est le suivant :

```
1 subgoals, subgoal 1 (ID 15)

  P : Prop
  Q : Prop
  R : Prop
  H : P -> Q
  H' : Q -> R
  p : P
  ============================
   R
```

Il y a un *but* à prouver : R, et le *contexte* comprend les termes P, Q, R, H, H' et p. Ainsi la tactique intros a introduit dans le contexte les propositions P Q R et les hypothèses H:P→Q, H':Q→R, p:P. Les noms donnés aux hypothèses sont attribués par Coq, mais nous pouvons les expliciter.

Désormais l'énoncé que l'on veut prouver est le même que la conclusion de l'hypothèse H'. Nous souhaitons donc appliquer cette hypothèse. La tactique apply prend un terme dont le type est une implication qui se termine par le même terme que le but, ou qui est le but lui même. L'application va éventuellement créer de nouveaux buts selon le terme appliqué. Dans notre exemple, en appliquant l'hypothèse H' le but courant est résolu, et le nouveau but est Q. L'étape suivante est d'appliquer l'hypothèse H. Le but à résoudre est maintenant P. Nous aurions pu de manière similaire aux cas précédents utiliser apply p, le but courant serait résolu et aucun nouveau but ne serait créé, la preuve serait terminée. Néanmoins lorsqu'une hypothèse correspond exactement à l'énoncé courant, nous pouvons utiliser dans ce cas la tactique assumption.

Chaque preuve doit se terminer par la commande Qed. Celle-ci va vérifier que le type de la preuve correspond à l'énoncé initial. Le terme de preuve peut être ensuite affiché comme n'importe quel terme de Coq grâce à la commande Print suivi du nom de l'énoncé :

```
Theorem imp_trans : ∀P Q R, (P →Q) →(Q →R) →P →R.
Proof.
  intros P Q R H H' p.
  apply H'. apply H. assumption.
Qed.
Print imp_trans.
```

Suite à cette dernière commande, Coq affichera :

```
imp_trans =
  fun (P Q R : Type) (H' : P -> Q) (H : Q -> R) (p : P) => H (H' p)
    : forall P Q R : Type, (P -> Q) -> (Q -> R) -> P -> R
```

Pour ce genre de preuves simples, Coq dispose de tactiques plus évoluées. Ainsi Proof. auto. Qed. permet également de prouver imp_trans.

Les preuves précédentes n'utilisent que des applications d'hypothèses. Examinons maintenant une preuve par induction. Lorsque nous définissons un type inductif, Coq génère automatiquement plusieurs théorèmes et fonctions pour travailler sur ces types,

dont un principe d'induction. Par exemple pour la liste définie plus haut, le principe d'induction est le suivant (la commande Check affiche le type d'une expression) :

```
list_ind  : forall (A : Type) (P : list A -> Prop),
       P (nil A) ->
       (forall (a : A) (l : list A), P l -> P (cons A a l)) ->
       forall l : list A, P l
```

Nous pouvons voir qu'en plus du paramètre A correspondant au type de la liste, nous avons une propriété P sur les listes. Ensuite les deux prémisses correspondent aux deux constructeurs. Le premier est classique et indique que la propriété est vérifiée pour la liste vide. Le second quant à lui, indique que la propriété doit être vérifiée pour la sous-liste. La conclusion indique que dans ce cas la propriété est vérifiée pour toute liste.

Examinons sur un exemple l'utilisation d'un tel principe d'induction. La propriété que l'on souhaite prouver est que pour tout indice inférieur à la taille de la liste, la fonction nth_error retourne nécessairement une valeur de type option dont le constructeur est Some (c'est à dire qu'une valeur est présente dans la liste à cette position).

```
Fact nth_error_lt_defined :
  ∀(A : Type) (l : list A) (k : nat), k < length l →(exists y, nth_error l k = Some y).
Proof.
  intro A. induction l; intros k H.
```

Après avoir mis le type A dans les hypothèses, nous effectuons l'induction sur la liste l, grâce à la commande induction l. Comme nous l'avons vu sur la définition de list_ind, il y a deux prémisses, il y a donc ici deux sous-buts à considérer.

Nous avons jusqu'à présent utilisé le symbole « . » après chaque tactique : c'est le symbole marquant la fin d'une commande. Mais on peut également combiner des tactiques. Lorsque les tactiques tac et tac', sont appliquées en séquence à un but b, cela signifie que la tactique tac est appliquée à b, et la tactique tac' à *l'ensemble* des sous-buts générés par la précédente tactique. On note tac; tac'. Il est possible de combiner plusieurs tactiques à condition de savoir à l'avance quels seront les buts générés. Dans notre exemple induction l génère deux sous-buts, et dans chacun de ces sous-buts, les hypothèses seront introduites grâce à la combinaison avec la tactique d'introduction. Cela permet une factorisation des tactiques, améliorant la compacité des scripts de preuves.

Dans un souci de lisibilité, Coq permet l'utilisation de « *bullets* ». Les symboles $-$, +, $*$ permettent de structurer les scripts de preuve. Ainsi si à un moment dans la preuve nous avons n sous-buts et $-$ est utilisé, alors les scripts de preuves pour tous ces n sous-buts doivent également être introduits par $-$. Ainsi les sous-buts générés lors de la preuve de l'un de ces sous-buts traitésont distingué des sous-buts initiaux. Les symboles + et $*$ peuvent être utilisés de même. Il est possible également d'utiliser { } pour marquer un bloc dans lequel il sera à nouveau possible d'utiliser les mêmes symboles pour structurer le script de preuve. Continuons la preuve :

```
– contradict H. simpl. omega.
```

Pour le cas de la liste vide, notre hypothèse H est k <length nil. Cette hypothèse n'est jamais vérifiée car k est un naturel, et la longueur de la liste vide est 0. Nous prouvons donc ce cas par contradiction. La commande contradict H, remplace le but courant par la négation de l'hypothèse H. La commande simpl effectue les réductions nécessaires pour transformer le but dans sa forme appropriée, ici elle transforme length nil en 0. La commande omega est utilisée pour résoudre des systèmes d'équations et d'inéquations linéaires sur les entiers et les naturels. Ici elle permet de conclure que k< 0 est impossible.

Le second but, correspond à la deuxième prémisse du principe d'induction list_ind. Nous avons donc l'hypothèse suivante pour une liste l :

```
IHl : forall k : nat, k < length l ->
                      exists y : A, nth_error l k = Some y
```

et nous devons le prouver sur une liste avec un élément de plus, c'est-à-dire a::l. Nous avons également comme hypothèse que la position k est dans cette liste.

```
– destruct k; simpl in *.
```

L'hypothèse d'induction nécessite une hypothèse sur la position que nous n'avons pas actuellement. Nous savons uniquement que la position k est inférieure à la taille de a::l. Il est donc nécessaire d'effectuer un raisonnement par cas sur la position. Cela se fait ici grâce à la commande destruct k qui génère deux sous-buts. Nous pouvons voir ensuite une variante de l'utilisation de simpl avec l'ajout de in *. Nous pouvons préciser grâce à in sur quelle hypothèse nous appliquons une tactique. Ici avec * nous l'appliquons sur toutes les hypothèses et sur le but courant.

Nous avons donc deux sous-buts. Le premier but correspond au cas k=0. Le but étant de démontrer qu'il existe une valeur dans la liste correspondant à cet indice (ici 0), il suffit de fournir le premier élément de liste, donc a. La commande exists suivie de la valeur remplit ce rôle. Nous terminons avec la commande trivial qui permet de conclure pour des égalités évidentes :

```
  + exists a. trivial.
```

Pour le second cas, l'hypothèse sur la position de l'indice k nous permet d'appliquer l'hypothèse d'induction et de conclure :

```
  + apply IHl. omega.
Qed.
```

Nous avons vu que le principe d'induction est généré automatiquement, néanmoins celui-ci peut s'avérer mal adapté aux raisonnements que l'on souhaite effectuer. Par exemple les traces de programmes sont des séquences d'événements. Elles sont construites à partir des différentes étapes de réduction d'un programme, où chaque nouvel événement est ajouté à la fin de la séquence. Une séquence est implantée par

une liste, néanmoins le principe d'induction généré par Coq n'est pas adapté. Le second constructeur de liste, crée cette nouvelle liste en ajoutant un élément en tête d'une liste existante. Le principe d'induction reproduit cette construction alors que nous souhaiterions avoir un ajout en fin de liste. Nous pouvons donc définir notre propre principe :

```
Lemma rev_ind :
  ∀P:list →Prop,
  P nil →
  (∀ (x:A) (l:list), P l →P (l ++ x :: nil)) →∀l:list, P l.
```

La preuve d'un tel principe se trouve dans le fichier `List` de Coq.

Nous pouvons également avoir besoin d'un principe d'induction généralisée. Sur notre exemple, cela signifie que la propriété ne doit pas être juste vérifiée pour la queue de la liste immédiate, mais pour toutes les listes de tailles strictement inférieures à la taille de la liste courante.

```
Lemma list_ind_length :
  ∀P : list →Prop,
  (∀ s, (∀ s', length s' < length s →P s') →P s) →
  ∀s : list, P s.
```

Il est possible en Coq de mêler valeurs « habituelles » que l'on trouve en programmation fonctionnelle et preuves de propriétés de ces valeurs, dans une seule valeur Coq. Pour ce faire, on peut utiliser le type suivant :

```
Inductive sig {A : Type} (P : A →Prop) : Type :=
  exist : ∀x : A, P x →sig P.
```

Notons l'utilisation d'accolades à la place de parenthèse pour le paramètre A : ce paramètre est *implicite*. Pour construite une valeur pour ce type, il faut donc une valeur a de type A et une valeur de type P a, c'est-à-dire une preuve que la valeur a possède bien la propriété P. Nous avons deux valeurs, on parlera donc de paire, et nous pouvons remarquer que le *type* de la seconde composante dépend de la *valeur* de la première composante : c'est un exemple de type dépendant, et ce genre de paire est appelée paire dépendante.

Par exemple on peut avoir :

```
Definition not_zero (x:nat) : Prop := x<>0.

Fact one_not_zero: not_zero 1. Proof. auto. Qed.

Definition one : sig not_zero := exist _ 1 one_not_zero.
```

Il existe néanmoins une notation pour ce type sig et on notera plutôt :

```
Definition one' : { n:nat | not_zero n} := exist _ 1 one_not_zero.
```

Cette notation met en relief le fait qu'on peut penser au type @sig A P [1] comme à un sous-type de A.

Il est également usuel de ne pas prouver des propriétés telles que one_not_zero avant de définir des valeurs telles que one. On utilisera plutôt l'une des deux formes suivantes.

Dans la première, nous faisons usage du mode de preuve interactive et de la tactique refine qui permet de donner un terme incomplet, les « trous » étant indiqués par le symbole _. Lorsque Coq n'est pas capable d'inférer le sous-terme manquant, il génère un sous-but. Dans cet exemple le sous-terme correspond à la preuve que la valeur donnée est différente de zéro : il est donc tout à fait commode d'être dans le mode de preuve pour définir ce sous-terme :

```
Definition one'' : { n:nat | not_zero n}.
Proof.
  refine (exist _ 1 _).
  auto.
Qed.
```

Dans la seconde, nous utilisons la fonctionnalité Program de Coq qui permet dans une définition de laisser des trous, qui donnent lieu à la génération d'*obligations de preuve* qui sont à prouver en mode interactif, le script de preuve de chaque obligation devant être introduit par la commande Next Obligation :

```
Program Definition one''' :
  { n : nat | n<> 0} := 1.
Next Obligation.
  auto.
Qed.
```

Dans cet exemple, si le module Program de Coq était chargée, l'obligation de preuve serait automatiquement prouvée et le système n'indiquerait aucune obligation à prouver.

Il est ainsi possible en utilisant des types sigma, de spécifier des pré-conditions sur les paramètres d'une fonction, et éventuellement une post-condition sur son résultat. Revenons à la définition de nth. Nous pouvons définir la fonction uniquement pour les indices inférieurs à la taille de la liste :

1. lorsqu'un terme f a des paramètres implicites, @f rend tous les paramères explicites.

```
Program Fixpoint nth2 {A:Type}(l:list A)(n:nat | n<length l) : A :=
  match l with
    | [] ⇒_
    | x::xs ⇒
      match n with
        | 0 ⇒x
        | S n ⇒nth2 xs n
      end
  end.
Next Obligation. simpl in *; omega. Qed.
Next Obligation. simpl in *; omega. Qed.
```

Ici (n:nat | n<length l) est un raccourci pour (n : {n:nat | n<length l}. Dans la définition de nth2 nous utilisons n comme si c'était une valeur de type nat. C'est la fonctionnalité Program qui se charge d'inclure les projections nécessaires. Dans l'appel récursif de nth2, le terme n n'est plus qu'un entier. Une obligation de prouver que cette valeur est inférieure à length l est donc générée. Le _ à droite du premier cas de filtrage sert de « trou », qui sera rempli par le terme de preuve produit en mode preuve lors de la preuve de l'obligation générée.

2.1.3 Modularité

Comme la plupart des langages de programmations, Coq permet d'organiser ses programmes en hiérarchies. Tout d'abord tout terme t d'un fichier `File.v` peut être accédé par l'expression File.t. On parle alors de la *bibliothèque* File. Il est possible de construire une hiérarchie plus profonde à l'aide de répertoires et d'options de compilation adéquates. Coq dispose également d'un système de modules. Cela améliore ainsi la réutilisabilité et la visibilité des notions définies. Ces modules sont divisés en deux parties. Les Module Type agissent comme une signature, ils définissent ce qui est visible pour le reste du programme. Ils peuvent donc être vus comme les *headers* en *C*, ou les interfaces de *Java*. Cependant il est possible d'ajouter des informations logiques. Les Module eux définissent une implantation.

Examinons l'exemple suivant, dans lequel nous voulons un type pour lequel l'égalité est décidable, et une fonction de comparaison où il existe un élément particulier qui est plus petit que tous les autres :

```
Module Type DecOrder.
  Parameter t : Set.
  Parameter eq_dec : ∀p p' : t, {p = p'} + {p <> p'}.
  Parameter lt : t →t →Prop.
  Parameter lt_irrefl : ∀p, ∼lt p p.
  Parameter lt_trans : ∀p p' p'', lt p p' →lt p' p'' →lt p p''.
  Parameter lt_asym : ∀p p', lt p p' →lt p' p →False.
  Parameter lt_dec : ∀p p' : t, {lt p p'} + {∼ lt p p'}.
  Parameter bot : t.
  Axiom bot_alway_lt : ∀(a:t), a <> bot →lt bot a.
End DecOrder.
```

Pour définir une signature nous commençons par les mot-clés Module Type. Les types et les opérations sont déclarés avec la commande Parameter, et les propositions avec la commande Axiom.

Les modules peuvent instancier un type de module de deux manières. Un module M peuvent instancier une signature S noté M:S, et alors toutes les définitions qui sont dans M et pas dans S ne seront pas visibles à l'extérieur. Un module M peut être compatible avec une signature S, noté M<:S, et les définitions supplémentaires seront visibles de l'extérieur. Les modules peuvent aussi être utilisés sans instancier aucune signature, afin de permettre de rassembler des définitions.

Dans la figure 2.2 nous implantons cette signature avec une liste de naturels. Il est nécessaire de donner une définition pour chaque paramètre et pour chaque axiome.

Égalité décidable. Dans le type de module DecOrderNat, nous pouvons voir une notation particulière utilisée dans la déclaration de eq_dec, le + : c'est le type prédéfini en Coq, sumbool. Sa définition, et la notation qui l'accompagne sont :

```
Inductive sumbool (A B:Prop) : Set :=
  | left : A →{A} + {B}
  | right : B →{A} + {B}
where "{ A } + { B }" := (sumbool A B).
```

On peut comprendre ceci comme étant une autre formulation de la disjonction, dont la définition est :

```
Inductive or (A B:Prop) : Prop :=
  | or_introl : A →A ∨ B
  | or_intror : B →A ∨ B
where "A ∨ B" := (or A B).
```

Outre la notation, la seule différence entre sumbool et or est le type : Set dans un cas et Prop dans l'autre. Tous les termes dans Coq sont typés, y compris les types eux-mêmes. En restant très informel, on peut dire que Set est le type des types qui correspondent aux types usuels des langages de programmation, ceux avec lesquels ont fait du calcul. Prop est le type des propriétés logiques. Le type de ces deux types

est Type. En fait il y a une suite infinie de types Type_i, mais sauf usage erroné très particulier, ceci n'est pas visible à l'utilisateur.

Une des grande différence entre Set et Prop est qu'on ne peut pas faire de filtrage de motif sur une valeur de Prop. Intuitivement ceci veut dire qu'on se soucie d'avoir des preuves, mais pas de comment elles ont été faîtes. En particulier on ne peut pas définir des fonctions dont le comportement serait dépendant de la façon dont une précondition d'un de ses arguments a été prouvée, ce qui semble tout à fait raisonnable.

Or l'égalité de deux termes est définie par :

```
Inductive eq (A:Type) (x:A) : A →Prop :=
  eq_refl : x = x
where "x = y" := (@eq A x y).
```

Là encore puisque que le type A est quelconque, c'est tout à fait raisonnable. En général il faudra faire une preuve de l'égalité de deux termes. Toutefois pour les types de données usuels utilisés en programmation, l'égalité est décidable. Pour chaque type, on peut prouver :

```
Theorem eq_dec : ∀x y, x = y ∨ x <> y.
```

Donc pour des x et y particuliers, l'application de ce théorème donne un terme construit soit avec or_introl soit avec or_intror. On voudrait ainsi traduire une conditionnelle usuelle `if n=0 then e1 else e2` en :

```
match (eq_dec n 0) with
 | or_introl _ ⇒e1
 | or_intror _ ⇒e2
end.
```

Toutefois (eq_dec n 0) est de type Prop, ce filtrage n'est donc pas autorisé. Par contre on peut également prouver le théorème :

```
Theorem eq_dec' : ∀x y, { x = y } + { x <> y }.
```

Puisque (eq_dec' x y) est de type Set, on peut filtrer une valeur de ce type et écrire :

```
match (eq_dec' n 0) with
 | left _ ⇒e1
 | right _ ⇒e2
end.
```

En fait pour les types bool et sumbool, Coq fournit une syntaxe usuelle :

```
if (eq_dec' n 0) then e1 else e2.
```

Foncteurs. Il est possible de paramétrer les modules avec d'autres modules. Ces modules paramétrés sont appelés des *foncteurs*. Sur l'exemple suivant, la structure de données Map associe une valeur à chaque valeur de clé. Pour que cette structure fonc-

```
Module DecOrderNat : DecOrder.

  Definition t := list nat.

  Definition eq_dec := List.list_eq_dec eq_nat_dec.

  Definition lt(p1 p2 : t) := exists p, p <> nil ∧ p1++p = p2.

  Definition lt_irrefl : ∀p, ∼lt p p.
  Proof. (* preuve omise *) Qed.

  Definition lt_trans : ∀p p' p'', lt p p' →lt p' p'' →lt p p''.
  Proof. (* preuve omise *) Qed.

  Definition lt_asym : ∀p p', lt p p' →lt p' p →False.
  Proof. (* preuve omise *) Qed.

  Definition lt_dec : ∀p p' : t, {lt p p'} + {∼ lt p p'}.
  Proof. (* preuve omise *) Qed.

  Definition bot := (nil : list nat).

  Lemma bot_alway_lt : ∀(a:t), a <> bot →lt bot a.
  Proof. intros a H. unfold lt, bot in *. exists a;auto. Qed.

End DecOrderNat.
```

FIGURE 2.2 – *Exemple de module Coq*

tionne l'égalité doit être décidable sur le type des clés, c'est pourquoi nous utilisons le module type DecOrder pour les représenter.

```
Module Type Map(X:DecOrder).
  Definition key:=X.t.
  Parameter t: Type.
  Parameter empty : t →t.
  Parameter add : key →t →(t →t ) →(t →t).
  Parameter find : key →(t →t) →option t.
End Map.
```

Nous avons fait un usage intensif du système de modules pour structurer nos développements.

2.2 Préservation sémantique et simulations

Soient $\mathcal{L}_S$ et $\mathcal{L}_T$ deux langages de programmation. On considère un compilateur $\mathcal{C}$ de $\mathcal{L}_S$ vers $\mathcal{L}_T$. Vérifier formellement ce compilateur revient à vérifier que la sémantique d'un programme de p est préservée par le programme $\mathcal{C}(p)$. Plus précisément les sémantiques formelles de $\mathcal{L}_S$ et $\mathcal{L}_T$ doivent permettre de définir les comportements observables des programmes de $\mathcal{L}_S$ et $\mathcal{L}_T$. La préservation sémantique la plus forte est d'exiger que les ensembles de comportements soient identiques : on parle alors de bissimulation. C'est une exigence trop forte pour un compilateur. Il est en effet usuel que les sémantiques de référence de langages de programmation soient sous-spécifiées, par exemple elles ne précisent souvent pas l'ordre d'évaluation des expressions, mais autorisent les compilateurs à faire un choix.

Pour prendre en compte ce cas de figure, on peut se contenter d'une simulation « en arrière » [55] : tous les comportements observables de $\mathcal{C}(p)$ sont des comportements observables de p. Enfin il n'est pas pertinent de s'intéresser à la préservation sémantique de programmes erronés. Si on note $safe(p)$ le fait qu'un programme ne peut avoir de comportement erroné, et qu'on note $obs_{\mathcal{L}}(p)$ l'ensemble des comportements observables de p pour la sémantique d'un langage $\mathcal{L}$, alors le résultat de préservation sémantique que l'on souhaite prouver est :

$$\forall p \in \mathcal{L}_S, safe(p) \Rightarrow \forall o \in obs_{\mathcal{L}_T}(\mathcal{C}(p)) \wedge o \in obs_{\mathcal{L}_S}(p).$$

Il est également possible pour les langages ayant une sémantique déterministe de prouver une simulation en avant (tous les comportements observables de p sont des comportements observables de $\mathcal{C}(p)$ [55]) : elle permet alors de prouver une simulation en arrière. Les simulations en avant sont plus faciles à prouver, mais ne peuvent être utilisées pour prouver la preuve de correction de passes de compilation de langages intrinsèquement non-déterministes. Ce type de simulation ne convient donc pas aux langages que nous allons traiter dans ce travail.

Les sémantiques opérationnelles que nous allons considérer sont des relations de la forme :

$$\vdash_p \Sigma \xrightarrow{e}_{\mathcal{L}} \Sigma'$$

où p dénote un programme, Σ et Σ' dénotent des états de programmes (l'état de la mémoire, c'est-à-dire le tas et les environnement locaux, l'état de ce qu'on pourrait considérer comme des compteurs ordinaux, ...), et enfin e dénote un évènement produit lors du passage de l'état Σ à l'état Σ'. Par abus de notation nous notons

$$\vdash_p \Sigma \xrightarrow{s}_{\mathcal{L}} \Sigma'$$

la relation où $s = \epsilon$ ou s est une trace constituée d'un seul évènement.

On distingue en général les états initiaux, et l'on note $\mathit{initial}^p_{\mathcal{L}}\ \Sigma$ pour indiquer que Σ est un état initial du programme p du langage $\mathcal{L}$, et les états finaux et l'on note $\mathit{final}^p_{\mathcal{L}}\ \Sigma$ pour indiquer que Σ est un état de terminaison normale du programme p. Un état final est tel que :

$$\forall \Sigma' \forall e,\ \neg\big(\vdash_p \Sigma \xrightarrow{e}_{\mathcal{L}} \Sigma'\big) \tag{2.1}$$

Les états qui vérifient (2.1) mais qui ne sont pas des états finaux sont des états d'erreur. On note alors $\mathit{stuck}^p_{\mathcal{L}}\ \Sigma$.

Notons s (respectivement $\underline{s}$) les séquences finies (respectivement infinies) d'évènements, et ϵ la trace vide. $s_1 s_2$ et $s_1 \underline{s_2}$ dénotent la concaténation d'une liste finie avec une liste finie (respectivement infinie). Suivant les notations de [55], on définit les clôtures transitives, réflexives et transitives des relations d'exécution et les réductions infinies par :

$$\frac{}{\vdash_p \Sigma \xrightarrow{\epsilon}{}^*_{\mathcal{L}} \Sigma} \tag{2.2}$$

$$\frac{\vdash_p \Sigma \xrightarrow{s_1}_{\mathcal{L}} \Sigma' \qquad \vdash_p \Sigma' \xrightarrow{s_2}{}^*_{\mathcal{L}} \Sigma''}{\vdash_p \Sigma \xrightarrow{s_1 s_2}{}^*_{\mathcal{L}} \Sigma''} \tag{2.3}$$

$$\frac{\vdash_p \Sigma \xrightarrow{s_1}_{\mathcal{L}} \Sigma' \qquad \vdash_p \Sigma' \xrightarrow{s_2}{}^*_{\mathcal{L}} \Sigma''}{\vdash_p \Sigma \xrightarrow{s_1 s_2}{}^+_{\mathcal{L}} \Sigma''} \tag{2.4}$$

$$\frac{\vdash_p \Sigma \xrightarrow{s_1}_{\mathcal{L}} \Sigma' \qquad \vdash_p \Sigma' \xrightarrow{\underline{s_2}}_{\mathcal{L}} \infty}{\vdash_p \Sigma \xrightarrow{s_1 \underline{s_2}}_{\mathcal{L}} \infty} \tag{2.5}$$

où une simple ligne indique une induction, et une double ligne une co-induction.

Nous pouvons alors considérer trois types de comportements pour un programme p, pour un état initial Σ^0

— $\vdash_p \Sigma^0 \xrightarrow{s}{}^*_{\mathcal{L}} \Sigma$ avec $\mathit{final}^p_{\mathcal{L}}\ \Sigma$, on note alors $p \Downarrow \mathit{converge}\ s\ \Sigma$,

— $\vdash_p \Sigma^0 \xrightarrow{\underline{s}}_{\mathcal{L}} \infty$, on note alors $p \Downarrow \mathit{diverge}\ \underline{s}$

— $\vdash_p \Sigma^0 \xrightarrow{s}{}^*_{\mathcal{L}} \Sigma$ avec $\forall \Sigma' \forall e,\ \neg(\vdash_p \Sigma \xrightarrow{e}_{\mathcal{L}} \Sigma')$ et $\neg \mathit{final}^p_{\mathcal{L}}\ \Sigma$, on note alors $p \Downarrow \mathit{goeswrong}\ s$.

On peut définir $\mathit{safe}(p)$ par $\forall o, p \Downarrow o \Rightarrow \forall s, o \neq \mathit{goeswrong}\ s$.

Pour prouver la préservation sémantique, il suffit de prouver la simulation à un pas, c'est-à-dire que pour tout programme p sûr, tout état $\Sigma_{\mathcal{L}_T}$ tel que

$$\vdash_p \Sigma_{\mathcal{L}_T} \xrightarrow{e}_{\mathcal{L}_T} \Sigma'_{\mathcal{L}_T},$$

et tout état $\Sigma_{\mathcal{L}_S}$ équivalent en un certain sens à $\Sigma_{\mathcal{L}_T}$ (on note $\Sigma_{\mathcal{L}_S} \sim \Sigma_{\mathcal{L}_T}$) alors il existe un état $\Sigma'_{\mathcal{L}_S}$ tel que

$$\vdash_p \Sigma_{\mathcal{L}_S} \xrightarrow{e}_{\mathcal{L}_S} \Sigma'_{\mathcal{L}_S} \text{ et } \Sigma'_{\mathcal{L}_S} \sim \Sigma'_{\mathcal{L}_T}$$

ce qu'on peut résumer sur le schéma suivant :

$$\begin{array}{lccc} p \vdash & \Sigma_{\mathcal{L}_S} & \xrightarrow{e} & \Sigma'_{\mathcal{L}_S} \\ & \wr & & \wr \\ p \vdash & \Sigma_{\mathcal{L}_T} & \xrightarrow{e} & \Sigma'_{\mathcal{L}_T} \end{array}$$

Lemme 2.1 (Préservation sémantique par simulation à un pas) *Si l'hypothèse précédente est satisfaite et la relation $\sim$ met en relation les états de même nature des deux langages, alors la compilation $\mathcal{C}$ préserve la sémantique des programmes sources sûrs.*

Démonstration. Soit p un programme sûr de $\mathcal{L}_S$, et soit o un comportement observable de $\mathcal{C}(p)$.

— Si $o = \mathit{converge}\ s\ \Sigma$. On raisonne par induction sur la définition de la clôture réflexive et transitive de la sémantique de $\mathcal{L}_T$.

— Si $o = \mathit{diverge}\ \underline{s}$. On raisonne par co-induction sur la définition de la divergence de la sémantique de $\mathcal{L}_T$.

— Si $o = \mathit{goeswrong}\ s$. Par hypothèse on a $\vdash_p \Sigma^0 \xrightarrow{s}{}^*_{\mathcal{L}_T} \Sigma$ et $\mathit{stuck}^p_{\mathcal{L}_T}\ \Sigma$. Par induction, on a $\vdash_p \Sigma^0_{\mathcal{L}_S} \xrightarrow{s}{}^*_{\mathcal{L}_S} \Sigma_{\mathcal{L}_S}$ et avec $\Sigma_{\mathcal{L}_S} \sim \Sigma$. Donc par propriété de $\sim$ on a $\mathit{stuck}^p_{\mathcal{L}_S}\ \Sigma_{\mathcal{L}_S}$. Mais ce n'est pas possible car p est sûr. Donc ce cas n'est pas possible.

□

La simulation à un pas est un hypothèse forte, et il est en pratique nécessaire de considérer des hypothèses plus faibles. En particulier pour notre passe de compilation, du code est rajouté pour manipuler des structures de données. L'exécution de ce code sera considérée comme produisant des évènements silencieux, qui ne seront pas pris en compte dans les traces.

La simulation option est définie par : pour tout programme p sûr, tout état $\Sigma_{\mathcal{L}_T}$ tel que $\vdash_p \Sigma_{\mathcal{L}_T} \xrightarrow{e}_{\mathcal{L}_T} \Sigma'_{\mathcal{L}_T}$, et tout état $\Sigma_{\mathcal{L}_S}$ équivalent à $\Sigma_{\mathcal{L}_T}$, soit on a la simulation à un pas, soit pour une mesure $|\cdot|$ et une relation $<$ bien fondée on a $|\Sigma'_{\mathcal{L}_T}| < |\Sigma_{\mathcal{L}_T}|$ et $\Sigma_{\mathcal{L}_S} \sim \Sigma'_{\mathcal{L}_T}$.

La seconde alternative de l'option simulation permet d'éviter que le programme compilé entre dans une boucle infinie d'évènements silencieux. Il est également possible de prouver la préservation sémantique à partir de la simulation option et de l'hypothèse sur la relation d'équivalence. Dans la suite du mémoire, nous nous intéresserons donc uniquement à la preuve de la simulation option pour nos passes de compilation.

État de l'art

3

Sommaire

Nos travaux s'inscrivent dans le cadre de la programmation concurrente. Dans un premier temps nous examinons rapidement la naissance de ce type de programmation.

La mémoire transactionnelle est l'une des solutions pour contrôler les accès concurrents à une mémoire partagée. Elle a permis de remettre en évidence le concept de section atomique. Nous examinons les implantations existantes de systèmes à transactions, en nous penchant plus particulièrement sur le traitement de l'emboîtement et des processus légers créés dans ces transactions. Les transactions ne sont pas le seul moyen de réaliser des sections atomiques. Nous nous intéressons également aux travaux existants sur l'inférence de verrous pour implanter de telles sections. Enfin nous étudions les travaux plus abstraitsxs à travers des sémantiques indépendantes de toute implantation.

Nous avons pour objectif de compiler notre langage à base de sections atomiques vers un langage à base de verrous. La correction de cette compilation doit être vérifiée. C'est-à-dire que nous devons établir un lien entre le comportement d'un programme source et d'un programme cible. Nous terminons donc ce chapitre en examinant des travaux sur la compilation certifiée.

3.1 Aux origines de la programmation concurrente

Les premières expériences sur la concurrence débutèrent avec l'utilisation des interruptions systèmes au début des années 1960 [13]. La programmation concurrente visait les systèmes d'exploitation mais il était reconnu que cela pouvait s'appliquer à un domaine plus vaste. Ces premiers développements ont été réalisés en assembleur

sans aucune conception formelle. Les erreurs dans les logiciels étaient inévitables dans ce cadre. Dijkstra, Brinch Hansen et Hoare commencèrent alors à réfléchir aux concepts fondamentaux de la concurrence.

Dijkstra [31] pose les bases de la programmation concurrente abstraite et fait l'hypothèse de l'indépendance de vitesse des processus. Chaque processus doit être considéré indépendamment, excepté pour les intercommunications explicites. Dijkstra énonce le concept des *sections critiques* comme étant une portion de code où un seul processus peut se trouver à la fois. Il définit les hypothèses pour établir les critères de corrections : exclusion mutuelle, équité et indépendance de vitesse. Des solutions théoriques existaient pour résoudre le problème de l'exclusion mutuelle, comme l'algorithme de Dekker, mais ces solutions n'étaient pas adaptées dans la pratique. Les concepteurs de matériel de leur coté ont résolu cela de manière drastique en désactivant les interruptions. Dijkstra propose les *sémaphores*, des variables accompagnées de deux primitives contrôlant l'accès à des sections que l'on souhaite protéger.

Hoare [46] cherche quant à lui à inventer de nouvelles abstractions de programmation pour le parallélisme. Avec pour objectif que les constructions proposées puissent permettre un grand nombre de vérifications statiques, il introduit : un composition parallèle d'instructions, une notion de ressource partagée par des processus parallèles, une construction de *region critique* dans laquelle une ressource partagée doit être utilisée de manière exclusive, et enfin une construction de région critique conditionnelle dans laquelle l'entrée est conditionnée par l'évaluation d'une condition booléenne.

Ces concepts ont été raffinés et mise en œuvre en particulier dans l'implantation de systèmes d'exploitation au cours des années 1970. Dans ces implantations, chaque processus a des ressources propres conséquentes, et sur un mono-processeur le passage de l'exécution d'un processus à un autre est coûteuse. Afin d'améliorer les performances et les communications, les processus légers (ou *thread*) ont été proposés. Ils permettent la concurrence à l'intérieur d'un même processus, en partagent les mêmes ressources, tout en ayant chacun leur propre pile, variables locales et compteur de programme. Ce concept gagna en popularité parmi la communauté UNIX dans les années 1980 [62]. Afin d'assurer la compatibilité, une version standardisée en *C* fut proposée pour faire son entrée dans POSIX. POSIX (*Portable Operating System Interface*) est un ensemble de standards spécifiés par IEEE, pour assurer la compatibilité entre les systèmes d'exploitations. Après plusieurs révisions, il fut accepté sous sa forme finale en 1996, et évolua ensuite. Cette *API* est composée des primitives pour la gestion (création,...), et la synchronisation (*mutexes*, signaux, ...) des processus légers.

3.2 Sections atomiques

Dans les systèmes de bases de données, un autre mécanisme de contrôle de la concurrence a été proposé par Gray [36]. Dans une transaction le code est exécuté en supposant qu'il n'y aura pas d'interférence avec les autres processus s'exécutant

en parallèle. Si une interférence a néanmoins lieu, les effets de la transactions sont annulés, et elle est exécutée de nouveau.

À la vue du succès des transactions dans les bases de données, une série de travaux s'est intéressée, au cours des années 80, à l'utilisation de systèmes transactionnels pour la programmation des systèmes distribués. Pour la mémoire partagée, le premier système [43] de mémoire transactionnelle repose sur les registres du processeur. Dans le premier système de mémoire transactionnelle, les *flags* des registres du processeur sont utilisés pour indiquer la validité ou non d'une valeur.

De nombreux systèmes de mémoire transactionnelle uniquement logiciels ont été conçus par la suite (section 3.2.1). Les transactions reposent donc sur l'hypothèse optimiste de non-interférence, au contraire des techniques à base d'inférence de verrous (section 3.2.2). Cette technique permet de bénéficier des avantages des verrous sans leurs inconvénients car ceux-ci ne sont pas gérés par l'utilisateur mais inférés par le compilateur. Enfin d'autres travaux ont considérés les sections atomiques d'un point de vue formel, en proposant des sémantiques indépendantes de toutes implantations (section 3.2.3).

3.2.1 Mémoire transactionnelle logicielle

Présenté comme une évolution du système précédent, les transactions se passent des registres pour devenir purement logicielles dans [80]. Le terme *Software Transactionnal Memory (STM)* est évoqué pour la première fois. Le système est statique, c'est-à-dire que les zones mémoires accédées par les transactions sont prédéterminées. Le système de transaction doit être non-bloquant, pour garantir que certaines transactions termineront. L'implantation consiste en une zone mémoire représentée par un vecteur de cellules, et d'un vecteur d'appartenance, qui associe à chaque cellule, une structure de données représentant la transaction. Cette représentation des transactions contient le numéro de version et un vecteur d'ancienne valeur des cellules. Le critère de correction est la *linéarisabilité* [44], c'est-à-dire toute trace d'exécution d'un programme doit être équivalente à une trace du même programme exécuté de manière séquentielle. Ces transactions seront utilisées pour implanter les régions conditionnelles critiques de Hoare [39].

La plupart des travaux suivants se basent également sur des transactions logicielles, mais certains [26] tentent de combiner les transactions logicielles et matérielles pour en faire un système hybride.

Pour améliorer la composabilité des transactions, Harris et al. proposent [40] l'ajout de nouvelles constructions comme la possibilité de spécifier une transaction à exécuter en cas d'échec de la première. L'implantation se base sur un journal stockant les différentes versions des cellules mémoires.

Un autre type d'implantation est proposé dans [34], en combinant un cache et des verrous. Pour accéder à une variable en écriture, le processus léger doit obtenir un verrou sur celle-ci. Les lectures sont gérées par un système de numéro de version.

Transactions emboîtées Ces précédentes implantations ne permettent pas les transactions emboîtées. Le besoin d'emboîtement est exprimé par Moss et al. dans [65], en pointant l'exemple de l'utilisation de bibliothèques, mais aussi par le fait que les sections atomiques sont moins monolithiques et donc plus légères. Cependant ils restreignent leur système en interdisant la concurrence à l'intérieur des sections atomiques mais en l'autorisant pour celles de plus haut niveau. Cette restriction vise à simplifier la détection de conflits et permet d'optimiser l'implantation.

La combinaison du parallélisme dans les sections et les sections atomiques emboîtées a été proposé par Agrawal et al. [4]. Ils ont choisi le parallélisme structuré à la place des *pthreads* habituellement utilisés, et ainsi ils se basent sur une représentation à base d'arbre pour la parallélisme et les sections atomiques. Chaque section atomique doit terminer avant que la section atomique parente termine.

Dans l'article [49] est présenté une sémantique d'un langage *Transactional Featherweight Java*, basé sur un noyau de *Featherweight Java* et gérant les transactions emboîtées. Deux implantations de la sémantique sont proposées. La première se base sur un journal des modifications de la transaction. La seconde, se base sur deux étapes de verrouillage : avant d'accéder à un objet, le verrou associé à celui-ci doit être acquis. Les verrous sont relâchés après la fin de la transaction.

Transactions dynamiques Dans l'article [42] est décrit un système de transaction dynamique, *Dynamic STM* (*DSTM*) pour Java. Avant cela l'utilisation mémoire et/ou les transactions devaient être connues à l'avance. *DSTM* se base sur des objets transactionnels accédés par des transactions. L'emboîtement de transactions n'est pas possible. Un objet transactionnel est un conteneur pour un objet Java. Pour qu'une classe soit encapsulable dans un objet transactionnel, elle doit posséder une méthode *clone*, car les transactions vont gérer plusieurs copies d'un objet. Cette classe possède trois champs, l'un stockant la transaction la plus récente qui a ouvert l'objet en écriture, un champ pointant sur la version de l'objet avant le début de la transaction et un champ pointant sur l'objet recevant les modifications. En cas de conflit l'ancienne version de l'objet sert de référence. En cas de succès la version en cours modifiée devient la nouvelle référence. L'un des aspects intéressants de ce travail est le fait de pouvoir en une étape, changer le statut de tous les objets ouverts en écriture par une même transaction par un simple basculement de pointeur. Ce système se présente sous la forme d'une bibliothèque, cette approche a été préférée à celle d'un compilateur, car celle-ci est plus facile à distribuer et à essayer.

Cette implantation a évolué dans la bibliothèque *DSTM2* [41]. *DSTM2* est une bibliothèque Java implantant les *STM* à base d'objet, corrigeant certains défauts de la précédente version. L'obligation de passer par un objet servant d'adaptateur impliquait que l'on ne devait pas accéder à l'objet original. De plus il est nécessaire de fournir une méthode de clonage pour chacun des objets transactionnels. Cela passe désormais par un interface à implanter pour ces objets.

Une solution sous forme d'un compilateur pour Java agrémenté d'instructions pour la gestion des transactions emboîtées est présentée dans [2]. La détection des

conflits peut s'effectuer à deux niveaux de granularité (objet et mot) : cela permet d'assurer à la fois un minimum de surcoût et d'avoir une bonne capacité de passage à l'échelle. Par exemple, il sera préférable pour un tableau d'utiliser le détection basée sur les mots. Les modifications se font en place. Un journal est maintenu pour pouvoir annuler les modifications. Les bénéfices se font sur les accès mémoire rapides, car il n'y a pas d'indirection et il n'y a pas de surcoût lié au clonage. Les bibliothèques existantes sur les objets sont utilisables, car il n'y a pas d'adaptateur sur les objets qui modifie leur type. Un *transaction record* traque l'état de chaque objet ou mot mémoire accédé dans une transaction. Cet enregistrement peut être dans deux états : partagé ou exclusif. Dans l'état partagé, il contient un numéro de version tandis que dans l'état exclusif, il contient un pointeur vers la transaction propriétaire. Chaque transaction maintient un ensemble de lecture, d'écriture, et un journal d'annulation. La sous-transaction utilise les ensembles de lecture, d'écriture et le journal de la transaction mère.

Partant du constat que les *STMs* introduisent un surcoût ralentissant l'exécution, les auteurs de l'article [3] proposent plusieurs optimisations, basées notamment sur l'analyse statique. L'implantation se base sur *Deuce* [51], un *framework* ajoutant à Java la gestion des transactions. Le choix du protocole des transactions avec ce *framework* est laissé à l'utilisateur. Deux techniques classiques d'optimisations à la compilation sont proposées. Ces techniques ne sont pas appliquées habituellement car en présence de concurrence elles ne seraient pas valides. Avec les *STMs* et leur propriété d'isolation, elles deviennent possibles. Par exemple certains chargements peuvent être éliminés. Si dans une boucle une même case d'un tableau est toujours lue à chaque tour, la lecture est reportée avant la boucle. L'analyse statique est utilisée pour améliorer l'implantation des transactions. Par exemple, avant chaque lecture d'une variable il est nécessaire de récupérer la valeur la plus récente. Si une analyse statique déduit que la variable n'a pas déjà été écrite depuis la dernière lecture, cette étape peut être évitée.

Serialisabilité des transactions Certains travaux se concentrent plus sur la vérification des propriétés offertes par les transactions. Dans [10, 52], les auteurs veulent établir la preuve de l'atomicité des transactions (sans emboîtement). Afin d'être abstrait, et ne pas être lié à une implantation de transaction spécifique, une sémantique basée sur les traces de programmes est proposée. Les preuves consistent à établir l'équivalence des traces des programmes avec une trace sérielle.

3.2.2 Inférence de verrous

Les sections atomiques ne sont pas toujours implantées à l'aide de transactions. Il existe également des travaux sur l'implantation à l'aide de verrous. Contrairement aux transactions qui supposent qu'il n'y aura pas d'interférences durant la section atomique, l'inférence de verrous adopte une approche pessimiste en empêchant les inférences de se produire. Les transactions présentent des problèmes de surcoût et gèrent difficilement les entrées/sorties. Les verrous doivent assurer l'atomicité, tout en maximisant la concurrence. Comme ces verrous sont ajoutés automatiquement, il

n'y a pas les problèmes liés à la gestion par l'utilisateur [38]. Ils sont également composables [61]. Les transactions nécessitent de l'instrumentation au moment de l'exécution, par exemple pour gérer les versions ou les journaux des modifications, tandis que l'inférence de verrous, cette instrumentation a lieu au moment de la compilation [25].

Dans [45], l'algorithme va essayer d'attribuer un verrou à chaque variable partagée et créer un ordre total sur cet ensemble de verrous pour éviter les interblocages. Ensuite au début de chaque section atomique, les verrous de chaque variable accédée dans cette section tentent d'être pris. Cet algorithme est ensuite optimisé pour minimiser le nombre de verrous en supprimant ceux qui sont inutiles. Par exemple, si pour toutes les sections atomiques, lorsque le verrou l_1 est pris l_2 l'est également, l_1 est inutile. La solution proposée dans [61] est similaire, mais moins évidente à utiliser, car dans le code source, des mutexes et des annotations sont nécessaires. Les performances de leur solution sont très proches de la solution où les verrous sont assignés manuellement, et meilleures que les solutions à base de transactions.

Ces précédentes approches partent des variables accédées dans la section atomique, pour déterminer un ensemble de verrous, et assigner cet ensemble ou un sous-ensemble à la section atomique. L'approche exposée dans [38] est plus une approche *top-down*. Les sections critiques et leurs interférences sont identifiées suite à différentes analyses effectuées sur la représentation sous forme d'un graphe des sections. Les verrous sont ensuite assignés pour protéger ces interférences. Tous les verrous sont pris avant que le premier ne soit relâché.

L'article [25] améliore cette inférence. L'analyse des verrous nécessaires à une section atomique, se fait en remontant le code en partant de la fin. Contrairement aux précédentes analyses, les chemins sont pris en compte, et pas uniquement les variables statiques. L'analyse se base sur une représentation à base de graphe pour pouvoir gérer les boucles et les branchements. Des verrous spécialisés en écriture et en lecture sont utilisés, pour améliorer la granularité. Enfin dans l'article [24], en prenant comme hypothèse que les objets partagés ne sont accédés qu'à l'intérieur des sections atomiques il est prouvé en Isabelle, que les objets accédés dans ces sections sont couverts par un verrou.

3.2.3 Sémantique des sections atomiques

Comme nous l'avons vu, de nombreuses implantations existent pour ces sections atomiques. Néanmoins certains travaux abordent ces sections d'un point de vue plus abstrait, à l'aide de sémantiques.

Jagannathan et al. fournissent une sémantique opérationnelle d'une version dérivée de Featherweight Java avec des sections emboîtées et des processus léger multiples, indépendante de toute implantation dans [49]. Ils permettent l'emboîtement, mais chacune des sections atomiques enfants doit terminer avant son parent. Leur travail sur la preuve de l'atomicité présentent quelques similarités avec le nôtre. Pour prouver la correction de leur travail, ils utilisent les traces de programme pour s'abstraire de la sémantique opérationnelle, et montrer la sérialisabilité, c'est-à-dire pour tout programme

sans annulation il doit y avoir une trace correspondante où les sections atomiques sont exécutées de manière sérielle.

Dans [64], Moore et al. mettent en évidence les problèmes communs découverts dans plusieurs implantations : le sens précis de l'atomicité et les limitations inutiles du parallélisme dans les sections atomiques. Pour éviter les problèmes liés au fait que le modèle d'atomicité (faible ou forte) ne soit pas clair, leur conseil est de donner une sémantique avec une preuve de l'atomicité. Ils l'illustrent à travers quatre langages, et l'un d'entre eux permet le parallélisme dans les sections atomiques avec deux primitives pour la création de processus léger. Le premier créé des processus légers qui doivent vivre entièrement à l'intérieur de la section parente, et donc ils ne peuvent créer que des sections internes. La seconde retarde la création à la fin de la section parente, et donc ces processus légers peuvent uniquement créer des sous-sections qui vivent à l'extérieur. Nous arrivons à la même conclusion pour les limitations inutiles, et nous allons plus loin en laissant les processus légers s'échapper des sections atomiques.

Dans [1], Abadi et al. veulent également faire face au problème lié à l'atomicité faible ou forte, avec le développement de transactions basées sur le modèle d'exclusion mutuelle automatique. Dans leur représentation, seule une section peut s'exécuter à la fois. Ils montrent que les implantations existantes peuvent mener à des exécutions surprenantes liées à l'atomicité faible. Ils présentent la sémantique de leur langage, et avec quelques restrictions ils peuvent donner le comportement de l'atomicité forte mais avec une implantation permissive.

3.3 Compilation certifiée

Une partie de notre travail consiste à compiler un langage à base de sections atomiques vers un langage à base de verrous. Pour garantir que les propriétés que l'on assure pour un programme source soient conservées dans le programme compilé, il est nécessaire d'avoir des preuves. La vérification de ces preuves par une machine permet de bénéficier d'un plus grande confiance dans celles-ci. Nous allons donc nous intéresser dans la suite à des travaux sur des compilateurs vérifiés formellement. Un compilateur est vérifié s'il a été formellement vérifié qu'il préserve la sémantique des programmes : pour tout programme source, si la compilation termine sans erreur, alors tous les comportements observables du code exécutable sont aussi des comportements observables du programme source.

Les premiers travaux sur la vérification de la compilation sont anciens. En 1967, McCarthy et Painter [60] vérifient qu'une compilation d'expressions arithmétiques vers un mini langage assembleur préserve la sémantique. Dans ce travail les variables sont directement traduites en accès à un environnement mémoire, et le processeur ne possède qu'un seul registre accumulateur. Bien que faîte sur papier, la preuve est très détaillée et l'objectif des auteurs était d'avoir *in fine* une preuve vérifiable par machine.

De nombreux autres travaux ont suivi, la bibliographie [29] cite cent publications de 1967 à 2003. Les travaux récents sur la compilation vérifiée de langages séquentiels comprennent : Jinja [50], des travaux plus spécifique sur l'automatisation de preuves [19] mais surtout les travaux autour de Compcert [56, 11, 54, 55].

Jinja est un langage de programmation à la Java. Il met en évidence les caractéristiques fondamentales du langage Java. [50] propose un modèle unifié pour le langage source, la machine virtuelle et le compilateur. Les preuves incluent la preuve de correction du compilateur et sont faites dans l'assistant de preuve Isabelle [68].

Le projet CompCert [1] explore la vérification formelle de compilateurs réalistes utilisables dans le logiciel embarqué. De tels compilateurs sont accompagnés de preuves mathématiques et vérifiés par la machine, que le code exécutable généré se comporte tel que décrit par la sémantique du programme source. La notion de préservation sémantique dans CompCert est une simulation en arrière : si le code *cible* s'exécute avec un comportement observable donné alors le code *source* également. Cependant pour prouver la simulation en arrière, l'approche de CompCert consiste à prouver la simulation en avant (qui est plus simple à prouver) et le déterministe du langage cible. Le compilateur CompCert est un compilateur optimisé pleinement vérifié de CompCert C (un large sous-ensemble de C) vers le code assembleur (actuellement ARM, PowerPc et x86-32) en Coq. De nombreux travaux très récents proposent d'étendre CompCert : prise en compte de l'arithmétique flottante [12], nouvelles passes de compilation [8]. Parmi les compilateurs basés sur le *back-end* de CompCert, on peut citer MLCompCert [27, 28] qui est un compilateur pour un langage séquentiel fonctionnel miniML qui se base sur le compilateur CompCert du langage intermédiaire CMinor.

Le travail sur les compilateurs vérifiés pour des langages parallèles est encore plus récent. Il concerne uniquement le parallélisme en *mémoire partagée*. Les extensions de CMinor [47] avec concurrence en mémoire partagée ne sont considérées que pour les programmes sans *data race*. Il n'y a pas encore de compilateur parallèle vérifié pour des telles extensions. [58] étend Jinja avec des primitives pour la création dynamique de processus légers, la synchronisation avec des moniteurs, le mécanisme *wait-notify* et la synchronisation de processus légers. [78, 77] étend CompCert Clight avec la création de processus légers et des primitives de lecture-modification-écriture atomiques, et considère la compilation uniquement sur les architectures x86 [79]. La preuve de correction du compilateur CompCertTSO ne peut pas utiliser la stratégie de preuve de CompCert de simulation en arrière utilisant la simulation en avant et le déterminisme du langage cible, car les langages cibles ne sont plus déterministes. Cependant les auteurs ont réussi à réutiliser des résultats de CompCert pour donner les résultats de la simulation en avant d'un seul processus léger en isolation et de combiner ces résultats pour prouver la simulation en arrière pour la composition de processus légers.

Notons dans les travaux en compilation vérifiée l'utilisation de l'expression modèle mémoire pour désigner deux modèles différents. Dans les travaux sur CompCert [56], un modèle mémoire abstrait et un modèle mémoire concret sont proposés comme formalisation de l'état mémoire d'une machine au cours de l'exécution d'un

1. `http://compcert.inria.fr`

programme C. La mémoire est modélisée comme une collection de blocs séparés. Dans la version concrète, un bloc est identifié par un entier naturel, et la mémoire est un tuple (N, B, F, C) où N désigne l'identifiant du prochain bloc pouvant être alloué, B les bornes du bloc, F une fonction indiquant si un bloc est désalloué ou non, et C une fonction qui associe une valeur à chaque bloc et décalage. Les opérations sur la mémoire ne changent pas la valeur de N, sauf l'allocation qui l'incrémente. Les identifiants de blocs sont donc uniques. Les auteurs précisent : « *Block identifiers are never reused, which greatly facilitates reasoning over "dangling pointers" (references to block previously deallocated).* » D'autre part le standard du langage C précise: « *The value of a pointer becomes indeterminate when the object it points to (or just past) reaches the end of its lifetime* ». Afin de formaliser ceci, la sémantique opérationnelle du langage CompcertC est indéfinie lorsque des pointeurs indéfinis sont manipulés, pas uniquement pour l'écriture ou la lecture, mais également pour le test de l'égalité de deux pointeurs. Les modèles mémoires que nous considérons sont plus simples que ceux de CompCert, néanmoins la prise en compte des références vers des blocs désalloués est nécessaire également. Nous discutons de cet aspect plus en détail dans la section 6.3.

Dans les travaux sur CompCertTSO [77], il est également question du modèle mémoire relaxé des processeurs de la famille x86. Il s'agit ici de décrire comme les opérations mémoires sont gérées par le processeur. Dans le travail présenté dans ce mémoire nous considérons un modèle mémoire « processeur » séquentiellement consistant.

Sections atomiques et processus légers : une définition formelle

4

Sommaire

Une première version du contenu de ce chapitre est parue dans : Frédéric Dabrowski, Frédéric Loulergue, Thomas Pinsard, Nested Atomic Semantics with Thread Escape: A Formal Definition*, ACM Symposium on Applied Computing (SAC), pages 1585–1592, ACM Press, Mars 2014.*

Avoir au sein d'un même langage la possibilité d'emboîter des sections atomiques et avoir du parallélisme interne à ces sections nécessite de précisément définir les notions de bonne synchronisation et d'atomicité. C'est l'objet de ce chapitre. Pour être indépendant d'un langage particulier nous travaillons sur des traces d'évènements pouvant être produites par des exécutions de langages impératifs de programmation ayant ces fonctionnalités. Le chapitre se termine par une description de la formalisation de cette étude dans l'assistant de preuve Coq.

4.1 Domaines sémantiques des traces

Dans ce chapitre, nous considérons un noyau de langages impératifs avec création de processus légers communicant via mémoire partagée. La synchronisation des accès

concurrents à cette mémoire est assurée grâce aux sections atomiques qui peuvent être emboîtées et supportent le parallélisme interne. Les processus légers créés à l'intérieur d'une section peuvent s'échapper de sa portée, c'est-à-dire qu'ils peuvent continuer leur exécution après la fin de la section. Nous affirmons que ce comportement asynchrone des processus légers, vis-à-vis des sections atomiques englobantes, bénéficie à la modularité des programmes. Ce choix contraste avec ceux de [64] où les processus légers doivent soit terminer complètement avant que la section termine, soit s'exécuter après la terminaison de la section (dépendant du choix de la primitive utilisée pour créer le processus léger).

Nous basons notre étude sur les traces partielles de programmes (section 4.1.1) et nous nous abstrayons de la syntaxe du programme en énonçant des propriétés de bonnes formations (section 4.1.2). En particulier, nous supposons que *l'atomicité faible*, c'est-à-dire la non-interférence des *sections atomiques concurrentes*, est assurée par le système de support à l'exécution à travers des mécanismes d'exclusion mutuelle. Puisque nous considérons du parallélisme interne et de l'échappement de processus léger, la définition d'interférence et de concurrence entre sections atomiques nécessite une attention spécifique. Un programme s'exécutant sans interférence dans une section atomique, signifie qu'il ne reçoit pas d'informations qui dépend de l'exécution de la section. Nous devons définir précisément quels processus légers et quelles sections atomiques doivent être considérés comme faisant partie de la section atomique.

4.1.1 Traces

Nous supposons des ensembles dénombrables d'emplacements en mémoire, de noms de processus léger et de noms de section, des éléments notés respectivement ℓ, t et $\mathfrak{s}$, éventuellement avec indices. L'ensemble de valeurs, dont les éléments sont notés v, éventuellement avec indices, contient au moins les emplacements mémoire, les entiers et les noms de processus léger.

Actions, évènements et traces. Nous définissons l'ensemble des actions, dont les éléments sont notés a, éventuellement avec indices, de la manière suivante :

$$\begin{array}{rcl} a & ::= & \mid \tau \\ & & \mid \texttt{alloc}\ \ell\ n \mid \texttt{free}\ \ell \mid \texttt{read}\ \ell\ n\ v \mid \texttt{write}\ \ell\ n\ v \\ & & \mid \texttt{fork}\ t \mid \texttt{join}\ t \mid \texttt{open}\ \mathfrak{s} \mid \texttt{close}\ \mathfrak{s} \end{array}$$

Intuitivement, τ dénote une action interne, non-observable. Une action $\texttt{alloc}\ \ell\ n$ dénote une allocation dans le tas d'un bloc de taille n à l'emplacement mémoire ℓ et une action $\texttt{free}\ \ell$ supprime un tel bloc du tas. Une action $\texttt{read}\ \ell\ n\ v$ (resp. $\texttt{write}\ \ell\ n\ v$) dénote une lecture (resp. écriture) de (resp. dans) l'emplacement ℓ avec un décalage de n et v est la valeur lue (resp. écrite). Les actions $\texttt{fork}\ t$ and $\texttt{join}\ t$ dénotent respectivement la création d'un processus léger et l'attente de terminaison d'un processus léger. Les actions $\texttt{open}\ \mathfrak{s}$ and $\texttt{close}\ \mathfrak{s}$ dénotent l'ouverture et la fermeture de sections. Notez

que le nom des sections est purement décoratif et n'a aucun contenu opérationnel. Cela sera formalisé dans les conditions de bonnes formations de la section 4.1.2. Leur unique but est de nommer les occurrences de sections atomiques ayant lieu dans les traces. Un *événement* e est un couple composé d'un nom de processus léger et d'une action, une *trace* s est une séquence finie d'événements.

$$\text{(events)} \quad e \quad ::= \quad (t, a) \qquad\qquad \text{(traces)} \quad s \quad ::= \quad \epsilon \mid s \cdot e$$

Notations On note $s_1 s_2$ la concaténation de traces partielles (ou traces en abrégé) s_1 et s_2 où par abus de notation e signifie $\epsilon \cdot e$. Pour $i \in \mathbb{N}$, nous définissons la fonction partielle π par $\pi_{es}(0) = e$ et $\pi_{es}(n+1) = \pi_s(n)$. Nous notons respectivement $\pi_s^{act}(i)$ et $\pi_s^{tid}(i)$ la première et la seconde projection sur un événement $\pi_s(i)$. Nous notons $e \in s$, s'il existe un i tel que $\pi_s(i) = e$, et par extension $a \in s$ si $\pi_s^{act}(i) = a$.

Définitions Pour définir précisément ce qui doit être considéré comme faisant partie d'une section atomique, nous introduisons des définitions auxiliaires. Étant donnée une trace s, les relations $owner_s$ et $father_s$ relient respectivement une section au processus léger propriétaire et un processus léger à son père. La relation $range_s$ dénote de la portée d'une section. Une section est dite pendante dans une trace si elle a été ouverte, mais pas fermée. Par définition une section $\mathfrak{s}$ s'étend jusqu'à la dernière position d'une trace s, si $\mathfrak{s}$ est pendante dans s. Pour une trace *bien-formée* s, définie dans la section 4.1.2, les relations $owner_s$, $father_s$ et $range_s$ définirons des fonctions partielles.

$$owner_s\ \mathfrak{s}\ t \triangleq (t, \texttt{open}\ \mathfrak{s}) \in s \qquad\qquad father_s\ t\ t' \triangleq (t', \texttt{fork}\ t) \in s$$

$$\frac{\pi_s^{act}(i) = \texttt{open}\ \mathfrak{s} \qquad \pi_s^{act}(j) = \texttt{close}\ \mathfrak{s}}{range_s\ \mathfrak{s}\ (i, j)} \qquad\qquad \frac{\pi_s^{act}(i) = \texttt{open}\ \mathfrak{s} \qquad \texttt{close}\ \mathfrak{s} \notin s}{range_s\ \mathfrak{s}\ (i, |s| - 1)}$$

Il est maintenant possible de définir précisément quels sont les processus légers et les sections atomiques pouvant être considérés comme faisant partie d'une section. Étant donné une section $\mathfrak{s}$ d'une trace s, la relation $tribe_s\ \mathfrak{s}$ est définie comme le plus petit ensemble de processus léger contenant le propriétaire de la section et les processus légers créés en tant qu'effet de bord de l'exécution de la section (relation $tribeChildren_s$).

$$\frac{range_s\ \mathfrak{s}\ (i, j) \qquad i < k \leq j \qquad owner_s\ \mathfrak{s}\ t' \qquad \pi_s(k) = (t', \texttt{fork}\ t)}{tribeChildren_s\ \mathfrak{s}\ t} \qquad \frac{tribeChildren_s\ \mathfrak{s}\ t' \qquad father_s\ t\ t'}{tribeChildren_s\ \mathfrak{s}\ t}$$

$$\frac{owner_s\ \mathfrak{s}\ t}{tribe_s\ \mathfrak{s}\ t} \qquad\qquad \frac{tribeChildren_s\ \mathfrak{s}\ t}{tribe_s\ \mathfrak{s}\ t}$$

Intuitivement, si t appartient à $tribe_s\ \mathfrak{s}$ alors le processus léger t fait partie du calcul de la section atomique $\mathfrak{s}$ et ainsi ne doit pas être considéré en tant que processus

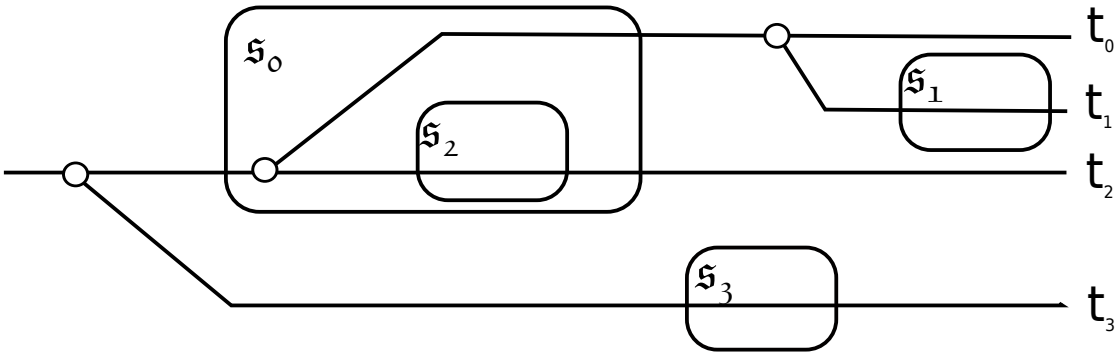


FIGURE 4.1 – *Exemple de processus légers et sections atomiques*

léger interférant. De manière similaire, nous définissons une relation sur les noms de sections, affirmant qu'une section atomique fait partie du calcul d'une autre. Nous disons que $\mathfrak{s}'$ est une *sous-section* de $\mathfrak{s}$ si $\mathfrak{s}' \Subset_s \mathfrak{s}$ tel que défini ci-dessous, est vérifiée.

$$\frac{\mathit{range}_s\ \mathfrak{s}\ (i,j) \qquad i < k \leq j \qquad \mathit{owner}_s\ \mathfrak{s}\ t \qquad \pi_s(k) = (t, \texttt{open}\ \mathfrak{s}')}{\mathfrak{s}' \Subset_s \mathfrak{s}}$$

$$\frac{\mathit{tribeChildren}_s\ \mathfrak{s}\ t \qquad \mathit{owner}_s\ \mathfrak{s}'\ t}{\mathfrak{s}' \Subset_s \mathfrak{s}} \qquad \frac{}{\mathfrak{s} \Subset_s \mathfrak{s}}$$

Enfin, deux sections atomiques sont dites concurrentes et noté $\mathfrak{s} \smile_s \mathfrak{s}'$, lorsque la condition (4.1) est vérifiée.

$$\mathfrak{s} \smile_s \mathfrak{s}' \triangleq \mathfrak{s} \notin_s \mathfrak{s}' \wedge \mathfrak{s}' \notin_s \mathfrak{s} \tag{4.1}$$

Exemple Examinons sur l'exemple de la figure 4.1 les différentes notions évoquées. Nous avons ici quatre processus légers t_0 à t_3 représentés par les traits horizontaux, et quatre sections atomiques $\mathfrak{s}_0$ à $\mathfrak{s}_3$ représentées par les rectangles. Les points blancs correspondent aux créations de processus légers. Soit s une trace issue de cet exemple. Nous avons $\mathit{owner}_s\ \mathfrak{s}_0\ t_2$ car t_2 initie cette section. Comme le processus léger t_2 crée t_0 nous avons également $\mathit{father}_s\ t_2\ t_0$. Le processus léger t_2 propriétaire de la section $\mathfrak{s}_0$ crée le processus léger t_0 sous la section atomique $\mathfrak{s}_0$, t_0 est donc un enfant de la tribu de $\mathfrak{s}_0$ ($\mathit{tribeChildren}_s\ \mathfrak{s}_0\ t_0$). Le processus léger t_1 l'est également car il est créé par un enfant de la tribu. La tribu de $\mathfrak{s}_0$ est donc constituée du propriétaire de la section t_2 et de ses deux enfants t_1 et t_0. La section $\mathfrak{s}_2$ est créée par le processus t_2 dans la section $\mathfrak{s}_0$, la section $\mathfrak{s}_2$ est donc une sous-section de $\mathfrak{s}_0$ ($\mathfrak{s}_2 \Subset_s \mathfrak{s}_0$). Comme t_1 est un enfant de la tribu de $\mathfrak{s}_0$, la section $\mathfrak{s}_1$ est une sous-section de $\mathfrak{s}_0$. Cela peut sembler inhabituel car la section $\mathfrak{s}_0$ est terminée lorsque la section $\mathfrak{s}_1$ commence, mais cela n'est du qu'à l'ordonnancement des processus légers. Un autre ordonnancement aurait pu faire que $\mathfrak{s}_1$ vive dans $\mathfrak{s}_0$. Les sections $\mathfrak{s}_0$ et $\mathfrak{s}_3$ sont quant à elles concurrentes. En effet t_3, propriétaire de la section $\mathfrak{s}_3$, est créé par t_0 avant le début de la section $\mathfrak{s}_0$, et n'est

donc pas un enfant de la tribu de $\mathfrak{s}_0$, nous avons donc $\mathfrak{s}_0 \smile_s \mathfrak{s}_3$ (t_2 n'est évidemment pas un enfant de la tribu de $\mathfrak{s}_3$).

4.1.2 Traces bien formées

Dans cette section, nous énonçons formellement des conditions de bonne formation sur les traces de programmes. Ces conditions peuvent être vues en tant que spécification pour de possibles implantations de langages avec sections atomiques emboîtées et échappement de processus léger. Elles vont de conditions de bon sens, à des choix de conception.

Pour formaliser ces conditions nous utilisons les définitions suivantes : le prédicat see_s peut être vu en tant que sur-approximation du flot d'informations dans s, et est défini en tant que clôture transitive de (4.2) ; Entre deux actions d'un même processus léger, il peut y avoir un échange d'informations via des variables locales. Entre la création d'un processus léger et les actions faites par ce processus, il peut y avoir un échange via l'argument donné à la création du processus. Enfin il peut y avoir un échange d'informations grâce à la lecture et l'écriture sur le tas. La relation $\prec_s$ sur les noms de section est définie en (4.3).

$$\frac{i < j \qquad \pi_s^{tid}(i) = t \qquad \pi_s^{tid}(j) = t}{see_s\ i\ j}$$

$$\frac{i < j \qquad \pi_s^{act}(i) = \texttt{fork}\ t \qquad \pi_s^{tid}(j) = t}{see_s\ i\ j} \tag{4.2}$$

$$\frac{i < j \qquad \pi_s^{act}(i) = \texttt{write}\ \ell\ n\ v \qquad \pi_s^{act}(j) = \texttt{read}\ \ell\ n\ v}{see_s\ i\ j}$$

$$\mathfrak{s} \prec_s \mathfrak{s}' \triangleq \exists i, j.\ i < j \wedge \pi_s^{act}(i) = \texttt{close}\ \mathfrak{s} \wedge \pi_s^{act}(j) = \texttt{open}\ \mathfrak{s}' \tag{4.3}$$

Une trace s est *bien formée* si elle satisfait les conditions de la Figure 4.2, expliquées ci-dessous :

- La condition ($\mathbf{wf}_1$) assure que les noms de section et de processus léger identifient respectivement les sections et les processus légers. Les conditions ($\mathbf{wf}_2$) et ($\mathbf{wf}_3$) énoncent de simples propriétés sur les noms de section. À chaque action de fermeture correspond une précédente action d'ouverture effectuée par le même processus léger. De plus, chaque action de fermeture d'un processus léger correspond à la dernière section ouverte, mais pas encore fermée, par le même processus léger. Ces conditions n'imposent aucune contrainte de gestion de noms sur l'implantation, car les noms de section sont purement décoratifs.
- Les conditions ($\mathbf{wf}_4$) et ($\mathbf{wf}_5$) énoncent les propriétés usuelles des instructions fork/join. La première énonce que les actions faites par un processus léger sont nécessairement faite après sa création. La seconde affirme que la création d'un processus léger et ses actions sont effectuées avant la synchronisation sur ce processus léger.

Toute action $\texttt{open}\ \mathfrak{s}, \texttt{close}\ \mathfrak{s}$ and $\texttt{fork}\ t$ apparaît au plus une fois dans s. $(\mathbf{wf_1})$

$$\forall i, \mathfrak{s}.\ \pi_s^{act}(i) = \texttt{close}\ \mathfrak{s} \Rightarrow \exists j.\ j < i \wedge \pi_s^{act}(j) = \texttt{open}\ \mathfrak{s} \wedge \pi_s^{tid}(i) = \pi_s^{tid}(j) \qquad (\mathbf{wf_2})$$

$$\begin{aligned}\forall \mathfrak{s},\ i,\ j.\ range_s\ \mathfrak{s}\ (i,j) \Rightarrow \pi_s^{act}(j) = \texttt{close}\ \mathfrak{s} \Rightarrow \\ \forall k,\ \mathfrak{s}'.\ i < k < j \Rightarrow \pi_s^{tid}(i) = \pi_s^{tid}(k) \Rightarrow \pi_s^{act}(k) = \texttt{open}\ \mathfrak{s}' \Rightarrow \\ \exists j',\ k < j' < j \wedge \pi_s^{act}(j') = \texttt{close}\ \mathfrak{s}'\end{aligned} \qquad (\mathbf{wf_3})$$

$$\forall i, t.\ \pi_s^{act}(i) = \texttt{fork}\ t \Rightarrow \forall j.\ \pi_s^{tid}(j) = t \Rightarrow i < j \qquad (\mathbf{wf_4})$$

$$\forall i, t.\ (\pi_s^{tid}(i) = t\ \vee\ \pi_s^{act}(i) = \texttt{fork}\ t) \Rightarrow \forall j.\ \pi_s^{act}(j) = \texttt{join}\ t \Rightarrow i < j \qquad (\mathbf{wf_5})$$

$$\forall \mathfrak{s}, i, j, t.\ range_s\ \mathfrak{s}\ (i,j) \Rightarrow owner_s\ \mathfrak{s}\ t \Rightarrow \forall k.\ \pi_s^{act}(k) = \texttt{join}\ t \Rightarrow j < k \qquad (\mathbf{wf_6})$$

$$\forall t, i, j.\ \pi_s^{act}(i) = \texttt{fork}\ t \Rightarrow \pi_s^{act}(j) = \texttt{join}\ t \Rightarrow see_s\ i\ j \qquad (\mathbf{wf_7})$$

$$\forall \mathfrak{s}, \mathfrak{s}',\ \texttt{open}\ \mathfrak{s} \in s \Rightarrow \texttt{open}\ \mathfrak{s}' \in s \Rightarrow \mathfrak{s} \smile_s \mathfrak{s}' \Rightarrow \mathfrak{s} \prec_s \mathfrak{s}' \vee \mathfrak{s}' \prec_s \mathfrak{s} \qquad (\mathbf{wf_8})$$

FIGURE 4.2 – *Conditions de bonne formation*

- La condition ($\mathbf{wf_6}$) énonce que la terminaison d'un processus léger ne peut être observée par un autre processus léger s'il a des sections pendantes. Une implantation peut au choix soit empêcher la terminaison d'un processus léger ayant des sections pendantes soit forcer la fermeture de telles sections à la terminaison. La condition ($\mathbf{wf_7}$) énonce qu'il n'est pas possible pour un processus léger d'attendre la terminaison d'un autre processus léger sans avoir explicitement reçu son nom. Ces conditions assurent qu'un processus léger externe n'interférera pas avec une section atomique en observant la terminaison des processus légers internes. Ces conditions correspondent à l'intuition que les sections atomiques devraient sembler être exécutées en un temps zéro et ainsi la terminaison de processus légers à l'intérieur d'une section ne devrait pas être observable avant que la section soit fermée. Les processus légers externes ne doivent pas voir la terminaison des processus légers interne pour éviter les interférences, mais cela aurait pu être effectué en interdisant l'opposé. Nous avons choisi de rendre l'intérieur d'une section invisible de l'extérieur plutôt que l'extérieur invisible depuis la section.
- La condition ($\mathbf{wf_8}$) énonce que les sections concurrentes ne se chevauchent pas.

Nous pouvons établir une relation pour une trace bien formée s entre les ouvertures de section dans la portée et la relation $\Subset_s$.

Lemme 4.1 *Pour toute trace bien formée s, section $\mathfrak{s}$ allant de i à j dans s et position k telle que $i \leq k \leq j$ et $\pi_s^{act}(k) = \texttt{open}\ \mathfrak{s}'$, pour un $\mathfrak{s}'$, nous avons $\mathfrak{s}' \Subset_s \mathfrak{s}$.*

Démonstration. Par hypothèse et par les conditions ($\mathbf{wf}_1$) et ($\mathbf{wf}_2$) nous n'avons ni $\mathfrak{s} \prec_s \mathfrak{s}'$ ni $\mathfrak{s}' \prec_s \mathfrak{s}$. Par ($\mathbf{wf}_8$) nous avons $\mathfrak{s} \Subset_s \mathfrak{s}'$ ou $\mathfrak{s}' \Subset_s \mathfrak{s}$. Supposons que $\mathfrak{s}$ et $\mathfrak{s}'$ sont ouvertes par deux processus légers distincts t et t' (sinon $i \leq k \leq j$ entraîne le résultat par la définition de $\Subset_s$) et $\mathfrak{s} \Subset_s \mathfrak{s}'$. Par définition de $\Subset_s$ nous avons $\mathit{tribeChildren}_s\ \mathfrak{s}'\ t$. Il est alors immédiat que $\pi_s^{act}(k) = \mathtt{open}\ \mathfrak{s}'$ et $\pi_s^{act}(i) = t$ impliquent $k < i$, contredisant ainsi notre hypothèse $\mathfrak{s} \Subset_s \mathfrak{s}'$. □

4.2 Traces bien synchronisées

L'identification du type d'isolation offerte par les sections atomiques est important. Traditionnellement deux types d'atomicité peuvent être distingués [59]. Dans la forme faible de l'atomicité les sections atomiques sont protégées seulement contre les autres sections. Cela signifie que les instructions à l'extérieur de la section peuvent interférer avec les données accédées dans les sections atomiques. Dans ce cas, les sections atomiques offrent uniquement une forme faible de protection. Au contraire, avec l'atomicité forte, le code à l'intérieur de section atomique est totalement protégé du code des autres sections et du code à l'extérieur de section.

Comme énoncé dans la section 4.1, nous comptons sur le système de support à l'exécution pour assurer certaines propriétés de l'atomicité faible : avec la condition ($\mathbf{wf}_8$) les sections concurrentes ne se chevauchent pas. Traditionnellement, nous définissons une notion de bonne synchronisation qui fournit une condition suffisante pour assurer l'atomicité forte (tel que définie dans la section 4.3). Pour ce faire nous définissons la notion de conflit sur les actions par la relation $\bowtie$ énoncée dans la figure 4.3 et affirmons qu'une trace est *bien synchronisée* si une synchronisation a lieu entre deux événements impliquant des actions conflictuelles. La synchronisation entre deux évènements est défini par *sw* qui est le plus petit prédicat définit par la fermeture transitive des règles de la figure 4.3.

Intuitivement, nous considérons un langage de programmation de haut niveau dans lequel la bonne synchronisation de devrait pas être vue comme une contrainte de programmation supplémentaire. En effet, dans de tels langages nous ne nous attendons pas à ce que l'utilisateur s'occupe des exécutions non séquentiellement consistantes. Il est responsable de l'écriture de programmes exempts de *data-race*. De plus les noms de processus léger et emplacement mémoire sont supposés être des valeurs de type de données opaque. En particulier, il doit y avoir une communication (et donc une synchronisation dans les traces bien-formées) entre l'allocation d'un emplacement (resp. la création d'un processus léger) et tout accès à cet emplacement (resp. join sur ce processus léger). Concernant les noms de processus léger, cela est assuré par la condition ($\mathbf{wf}_7$). Nous n'imposons pas une condition équivalente pour les emplacements mémoires ici, car cela est inutile dans notre cas, mais en pratique nous considérerons des programmes satisfaisant une telle propriété.

$$\begin{array}{rcl} \texttt{read}\ \ell\ n\ v & \bowtie & \texttt{write}\ \ell\ n\ v' \\ \texttt{write}\ \ell\ n\ v & \bowtie & \texttt{read}\ \ell\ n\ v' \\ \texttt{write}\ \ell\ n\ v & \bowtie & \texttt{write}\ \ell\ n\ v' \end{array}$$

$$\frac{i < j \quad \pi_s^{tid}(i) = t \quad \pi_s^{tid}(j) = t}{sw_s\ i\ j}$$

$$\frac{i < j \quad \pi_s^{act}(i) = \texttt{fork}\ (t) \quad \pi_s^{tid}(j) = t}{sw_s\ i\ j}$$

$$\frac{i < j \qquad \pi_s^{tid}(i) = t \qquad \pi_s^{act}(j) = \texttt{join}\ (t)}{sw_s\ i\ j}$$

$$\frac{i < j \qquad \pi_s^{act}(i) = \texttt{close}\ \mathfrak{s} \qquad \pi_s^{act}(j) = \texttt{open}\ \mathfrak{s}' \qquad \mathfrak{s} \smile_s \mathfrak{s}'}{sw_s\ i\ j}$$

FIGURE 4.3 – *Actions conflictuelles et prédicat « synchronisé avec »*

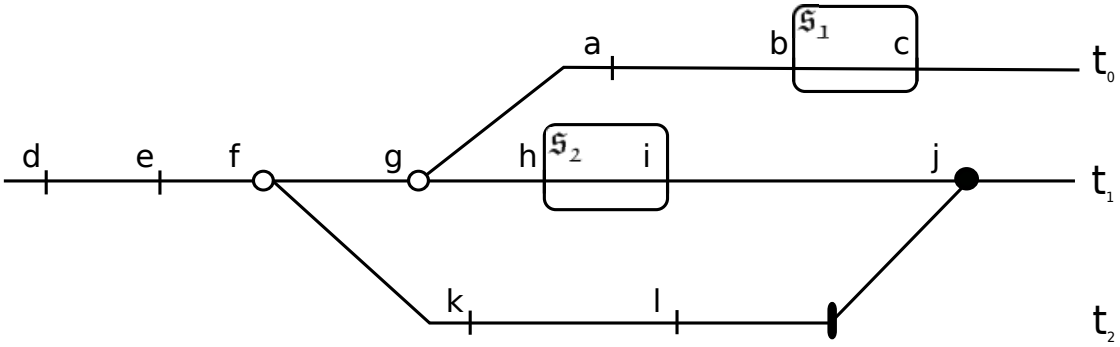


FIGURE 4.4 – *Exemples de synchronisation*

Exemple Examinons sur l'exemple présenté sur la Figure 4.4 quelques synchronisations entre actions. Les notations utilisées sont les mêmes que celles de l'exemple 4.1. Les points noirs correspondent aux actions de synchronisation de processus légers. Ici chaque action est étiquetée avec une lettre. Les quatre exemples suivants illustrent chacun une possibilité de synchronisation présentée dans la Figure 4.3. La première action de chaque couple, précède toujours la seconde dans la trace, condition nécessaire pour la synchronisation.

— d et e : elles sont effectuées par le même processus léger
— f et k : l'action f crée un processus léger qui effectue l'action k
— l et j : l'action l est effectuée par un processus léger qui est synchronisé par l'action j

— i et b : i correspond à la fermeture et b à l'ouverture de deux sections concurrentes.
— d et k : par transitivité

Nous pouvons maintenant définir notre propriété de bonne synchronisation, en exigeant que chaque action en conflit soit en synchronisation.

Définition 4.1 *Une trace s est bien synchronisée si pour toutes actions en conflits a et a' ayant lieu respectivement à la position i et j dans s tel que $i < j$, nous avons $sw_s\ i\ j$.*

Le lemme suivant énonce que dans une trace bien synchronisée, l'information ne peut circuler sans synchronisation.

Lemme 4.2 *Pour toute trace bien synchronisée s nous avons $see_s\ k\ k' \Rightarrow sw_s\ k\ k'$ pour tout k et k'.*

Démonstration. Immédiat par induction sur la preuve de $see_s\ k\ k'$. □

Exemples Nous nous attendons, dans une trace à la fois bien formée et bien synchronisée, qu'une section n'ait aucune interférence avec l'extérieur. Cela motive les conditions de bonne formation ($\mathbf{wf}_6$) et ($\mathbf{wf}_7$). Sans ces conditions, les traces de la figure 4.5 seraient bien formées et bien synchronisées mais auraient en fait un problème d'interférence. Les points blancs et noirs dénotent respectivement les actions *fork* et *join*. Les sections atomiques sont symbolisées par les boites arrondies. Les lettres "w" et "r" représentent respectivement les opérations d'écritures et de lecture sur le même emplacement mémoire donné (et nous supposons que cet emplacement mémoire n'est pas utilisé ailleurs dans la section $\mathfrak{s}$). Dans les deux cas l'opération d'écriture rouge interfère avec l'action de lecture sur l'emplacement mémoire dans la section $\mathfrak{s}$. Sans l'action `join` faite par le processus léger t_0, les deux écritures en conflit ne seraient pas synchronisées. La condition ($\mathbf{wf}_6$) interdit le `join` dans le premier cas, et la condition ($\mathbf{wf}_7$) l'interdit dans le second.

Nous pouvons maintenant lier notre notion de section et de synchronisation pour exprimer une propriété sur les tribus. La proposition suivante énonce que dans les traces bien synchronisées, les membres de la tribu ne peuvent pas se synchroniser avec des non membres tant que la section est active. Un corollaire immédiat est que dans les traces bien synchronisées, les actions des membres de la section ne peuvent être en conflit avec des actions des non membres tant que la section est active.

Proposition 4.1 *Pour toutes trace bien formée et bien synchronisée s, section $\mathfrak{s}$ allant de i à j dans s, et k, k' tel que $i \leq k, k' \leq j$ et $sw_s\ k\ k'$ nous avons $tribe_s\ \mathfrak{s}\ t \Rightarrow tribe_s\ \mathfrak{s}\ t'$ où $\pi_s^{tid}(k) = t$ et $\pi_s^{tid}(k') = t'$.*

Démonstration. Soit $\mathfrak{s}$ une section allant de i à j dans s et soit k' tel que $i \leq k' \leq j$ et

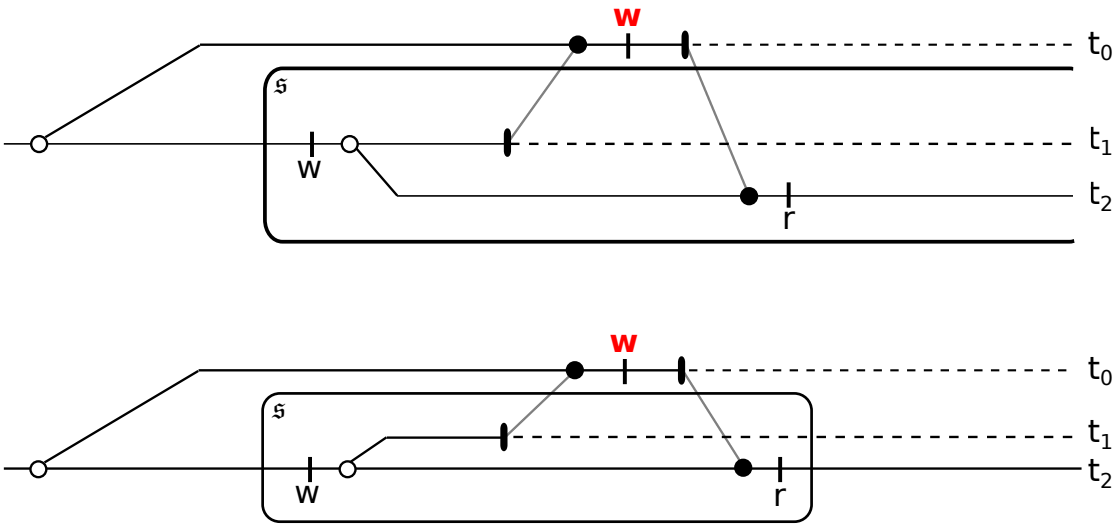


FIGURE 4.5 – *Deux exemples de traces mal formées*

$\pi_s^{tid}(k') = t'$ pour tout t'. Nous considérons la propriété:

$$P(k_0) \equiv \forall k, t'. \quad i \leq k_0 \leq j \wedge i \leq k \leq j \wedge sw_s\ k\ k_0 \wedge \pi_s^{tid}(k) = t \wedge \pi_s^{tid}(k_0) = t' \wedge tribe_s\ \mathfrak{s}\ t \Rightarrow tribe_s\ \mathfrak{s}\ t'$$

Supposons maintenant que $P(k_0)$ soit vérifiée pour tout $k_0 < k'$. Nous prouvons que $P(k')$ est vérifiée par induction sur la dérivation de la fermeture transitive de $sw_s\ k\ k'$.

- Si $\pi_s^{tid}(k) = \pi_s(k')$ ou $\pi_s^{act}(k) = \texttt{fork}(t')$ le résultat est immédiat par définition de *tribe*.
- Supposons que $\pi_s^{act}(k') = \texttt{join}(t)$. Par ($\mathbf{wf_6}$) nous avons $tribeChildren_s\ \mathfrak{s}\ t$ et ensuite, par définition de *tribe*, nous pouvons trouver un k_0 et un t_0 tel que $i < k_0 < k'$, $tribe_s\ \mathfrak{s}\ t_0$ et $\pi_s^{act}(k_0) = \texttt{fork}(t)$. Par ($\mathbf{wf_7}$) nous obtenons $see_s\ k_0\ k'$. Par définition de *see*, il est facile de vérifier que $\pi_s^{act}(k') = \texttt{join}(t)$ implique qu'il existe un $k_1 < k'$ tel que soit $\pi_s^{tid}(k_1) = t'$ ou $\pi_s^{act}(k_1) = \texttt{fork}(t')$ et soit $k_0 = k_1$ ou $see_s\ k_0\ k_1$. Si $k_0 = k_1$, le résultat est immédiat. Sinon, $P(k_1)$ est vérifiée par hypothèse d'induction, et par le lemme 4.2 nous avons $sw_s\ k_0\ k_1$. Par définition de *tribe* nous obtenons $tribe_s\ \mathfrak{s}\ t'$.
- Supposons que $\pi_s^{act}(k) = \texttt{close}(\mathfrak{s}')$ et $\pi_s^{act}(k') = \texttt{open}(\mathfrak{s}'')$ pour un $\mathfrak{s}'$ et un $\mathfrak{s}''$. Le résultat est immédiat par le lemme 4.1.
- L'étape d'induction est immédiate en appliquant deux fois l'hypothèse d'induction.

□

4.3 Atomicité

Dans cette section, nous prouvons que les traces bien formées et bien synchronisées satisfont la propriété d'atomicité forte. Plus formellement, nous prouvons que les traces bien synchronisées sont *serialisables*, c'est-à-dire de telles traces sont *équivalentes* à une trace *sérielle*. Une trace sérielle [72], est traditionnellement définie en tant que trace obtenue par un programme où les sections atomiques sont exécutées en série, c'est-à-dire sans entrelacement. Cette définition est bien adaptée lorsqu'il n'y a pas de parallélisme interne. Néanmoins dans notre cas, un entrelacement de processus légers « autorisés » est possible durant l'exécution de la section. C'est pourquoi nous devons définir notre propre notion de sériabilité. Pour définir cette notion de trace sérielle, nous devons prendre en compte le fait que les sections supportent l'emboîtement, le parallélisme interne et l'échappement de processus léger.

C'est exactement le but de la notion de tribu qui capture l'ensemble des processus légers qui devraient être autorisés à s'exécuter pendant que la section est active.

Définition 4.2 *Une trace s est* sérielle *si pour toute section $\mathfrak{s}$ et positions i, j et k tels que $range_s\ \mathfrak{s}\ (i,j)$ et $i \leq k \leq j$ nous avons $tribe_s\ \mathfrak{s}\ t$ où $\pi_s^{tid}(k) = t$.*

Définition 4.3 *Une trace s est sérialisable s'il existe une trace sérielle s' équivalente à s.*

Nous avons besoin maintenant de définir formellement ce que signifie pour deux traces d'être équivalentes. Habituellement, les traces sont définies étant équivalentes à un ré-ordonnancement préservant l'ordre des actions en conflits. Pour des raisons techniques, nous considérons une définition plus forte de l'équivalence, basée sur la synchronisation, mieux adaptées à nos travaux. Cette synchronisation est plus forte car elle implique la définition habituelle sur les conflits.

Définition 4.4 *Deux traces s et s' sont équivalentes, notées $s \equiv s'$, s'il existe un bijection γ entre les positions de s et s' tel que*

$$\pi_s(i) = \pi_{s'}(\gamma(i)) \qquad \text{pour tout } i < |s| \qquad (\mathbf{e_1})$$

$$sw_s\ i\ j \Leftrightarrow sw_{s'}\ \gamma(i)\ \gamma(j) \qquad \text{pour tout } i, j < |s| \qquad (\mathbf{e_2})$$

Proposition 4.2 *Pour toute trace bien formée et bien synchronisée s, et pour toute trace s', si $s \equiv s'$ alors s' est bien formée et bien synchronisée.*

Démonstration. La preuve de bonne synchronisation est faite en examinant l'ordre entre les positions dans les deux traces et en exploitant les propriétés de la bijection pour montrer que ces positions sont toujours en synchronisation. La preuve de bonne formation est faite pour chacune des huit conditions, en utilisant les propriétés de la bijection. □

Théorème 4.1 *Toute trace bien formée et bien synchronisée est sérialisable.*

Démonstration. La preuve est faite par induction structurelle sur s. Le résultat est évident pour la trace vide. Supposons que $s \cdot (t, a)$ est une trace bien formée et bien synchronisée. Alors s est à la fois bien formée et bien synchronisée car ces propriétés sont closes par préfixe. Par l'hypothèse d'induction, il existe s' tel que $s' \equiv s$ et s' est sérielle.

Ensuite nous prouvons que $s' \cdot (t, a) \equiv s \cdot (t, a)$. La bijection utilisée (notée γ) est la même que celle dans $s \equiv s'$ pour toutes les positions inférieures strictement à $|s|$ et $\gamma(|s|) = |s'|$. La preuve de la condition ($\mathbf{e_1}$) de l'équivalence est triviale. La partie la plus difficile de la condition ($\mathbf{e_2}$) advient lorsque nous avons $sw_{s\cdot(t,a)}\ i\ |s|$, donc nous avons à prouver $sw_{s'\cdot(t,a)}\ \gamma(i)\ \gamma(|s|)$. Nous savons que $\pi_s(i) = \pi_{s'}(\gamma(i))$ et $\pi_{s\cdot(t,a)}(|s|) = \pi_{s'\cdot(t,a)}(\gamma(|s|)) = (t, a)$. Par une induction sur $sw_{s\cdot(t,a)}\ i\ |s|$ nous pouvons conclure.

Par la proposition 4.2, $s' \cdot (t, a)$ est bien formée et bien synchronisée.

Maintenant montrons que $s' \cdot (t, a)$ est sérialisable. Nous notons $excludes_{s,\mathfrak{s}}\ t$ quand $\mathfrak{s}$ est pendante dans s et $\neg tribe_s\ \mathfrak{s}\ t$. Supposons que $\mathfrak{s}$ est la section la plus à gauche telle que $excludes_{s',\mathfrak{s}}\ t$ soit vérifiée.

— Supposons qu'une telle section n'existe pas. Soit les positions i, j, k, le nom de processus léger t' et le nom de section p' tel que $range_{s'\cdot(t,a)}\ \mathfrak{s}'\ (i, j)$, $\pi^{tid}_{s'\cdot(t,a)}(k) = t'$ et $i \leq k \leq j$. Nous voulons prouver que $tribe_{s'\cdot(t,a)}\ \mathfrak{s}'\ t'$. Si $k < |s'|$ le résultat est immédiat par la seriabilité de s'. Nous supposons maintenant que $k = |s'|$ (et donc $t = t'$). Si $\pi^{act}_{s'\cdot(t,a)}(k) = \texttt{open}\ \mathfrak{s}'$ alors $k = i$ par ($\mathbf{wf_1}$), et donc en tant que propriétaire de la section p', t' est dans la tribu de p'. Si $\pi^{act}_{s'\cdot(t,a)}(k) \neq \texttt{open}\ \mathfrak{s}'$ alors nécessairement p' est pendante dans s'. Par hypothèse nous avons ainsi nécessairement que $tribe_{s'}\ \mathfrak{s}'\ t'$ est vérifiée. Comme les tribus sont préservées par la concaténation de traces on a $tribe_{s'\cdot(t',a)}\ \mathfrak{s}'\ t'$: la trace $s' \cdot (t, a)$ est sérielle.

— Si la section $\mathfrak{s}$ existe, soit la position i telle que $\pi^{act}_{s'}(i) = \texttt{open}\ \mathfrak{s}$. Nous raisonnerons en utilisant l'insertion qui est utile pour représenter le décalage d'un élément dans la trace. Étant donnée une trace non vide $s \cdot e$, nous notons $s \leftarrow_{i_0} e$ la trace obtenue en insérant l'évènement e dans la trace s à la position i_0. Une bijection γ est induite par cette insertion:

$$\gamma(k) = k \text{ si } k < i \qquad \gamma(k) = k + 1 \text{ si } i \leq k < |s| \qquad \gamma(|s|) = i_0$$

Notez que γ conserve l'ordre relatif des positions inférieures à $|s|$ et leur inverse pour les positions différentes de i_0. Il y a deux propriétés d'insertion

importantes que nous utilisons:

Pour toutes traces bien formées et bien synchronisées $s \cdot (t, a)$,
pour tout nom de section $\mathfrak{s}$ et position i tel que
- $\mathit{excludes}_{s,\mathfrak{s}}\ t$ et $\pi_s^{act}(i) = \texttt{open}\ \mathfrak{s}$
- $\forall k.\ i \leq k < |s| \Rightarrow \neg sw_{s\cdot(t,a)}\ k\ |s|$

(ins$_{\mathbf{eq}}$)

nous avons $s \cdot (t, a) \equiv s\ {}^{\curvearrowleft}|_i\ (t, a)$

Pour toute trace bien formée $s \cdot (t_0, a)$ et position i telles qu',
- il existe un nom de section $\mathfrak{s}_0$ tel que $\mathit{excludes}_{s,\mathfrak{s}_0}\ t_0$
- $\forall k.\ i \leq k < |s| \Rightarrow \neg sw_{s\cdot(t_0,a)}\ k\ |s|$

(ins$_{\mathbf{tribe}}$)

alors pour tout $\mathfrak{s}$ et t, $\mathit{tribe}_{s\cdot(t_0,a)}\ \mathfrak{s}\ t \Rightarrow \mathit{tribe}_{s{}^{\curvearrowleft}|_i(t_0,a)}\ \mathfrak{s}\ t$

Considérons la trace $s'' = s'\ {}^{\curvearrowleft}|_i\ (t, a)$. Nous prouvons maintenant que s'' est équivalente à $s \cdot (t, a)$, et sérielle. Pour le faire nous avons besoin d'utiliser deux propriétés de l'insertion, donc premièrement nous prouvons $\forall k,\ i \leq k < |s'| \Rightarrow \neg sw_{s'\cdot(t,a)}\ k\ |s'|$. Supposons $sw_{s'\cdot(t,a)}\ k\ |s'|$ pour un k telle que $i \leq k < |s'|$. Par la seriabilité de s' nous savons que $\mathit{tribe}_{s'}\ \mathfrak{s}\ t_1$, où $\pi_s^{tid}(k) = t_1$, et $\mathit{tribe}_{s'\cdot(t,a)}\ \mathfrak{s}\ t_1$ par préservation de *tribe* par concaténation. Par la proposition 4.1, nous pouvons conclure que $\mathit{tribe}_{s'\cdot(t,a)}\ \mathfrak{s}\ t$. Comme $s' \cdot (t, a)$ est bien formée nous savons que $a \neq \texttt{open}\ \mathfrak{s}$ et $a \neq \texttt{fork}\ t$. Par définition de *tribe* nous obtenons que $\mathit{tribe}_{s'}\ \mathfrak{s}\ t$ ce qui contredit notre hypothèse $\neg\mathit{tribe}_{s'}\ \mathfrak{s}\ t$.

Par (**ins$_{\mathbf{eq}}$**) et transitivité, nous concluons que $s'' \equiv s \cdot (t, a)$.

Pour prouver que s'' est sérielle, soit les positions i', j', k, le nom de section $\mathfrak{s}$, le processus léger t tel que $\mathit{range}_{s''}\ \mathfrak{s}'\ (i', j')$, $i' \leq k \leq j'$ et $\pi_{s'}^{tid}(k) = t'$. Pour prouver $\mathit{tribe}_{s''}\ \mathfrak{s}'\ t'$ nous prouvons premièrement $\mathit{tribe}_{s'}\ \mathfrak{s}'\ t'$ en transposant le $\mathit{range}_{s''}\ \mathfrak{s}'\ (i', j')$ dans s' et nous utilisons la seriabilité de s' pour conclure. Pour ce faire, prouvez que les position relatives de i', k et j' sont préservées par l'inverse de la bijection induite γ, c'est-à-dire considérez si chaque position est égale à i ou pas: premièrement prouvez que $i' \neq i$ et concluez pour les quatre cas restant en utilisant les conditions de bonne formation. Donc $\mathit{tribe}_{s'}\ \mathfrak{s}'\ t'$ est vérifiée.

Comme *tribe* est préservée par la concaténation de trace, nous avons $\mathit{tribe}_{s'\cdot(t,a)}\ \mathfrak{s}'\ t'$ et par (**ins$_{\mathbf{tribe}}$**), nous concluons $\mathit{tribe}_{s''}\ \mathfrak{s}'\ t'$. Donc s'' est sérielle.

□

4.4 Formalisation en Coq

4.4.1 Organisation générale

Nous avons formalisé le contenu de ce chapitre dans l'assistant de preuve Coq. Cela a nécessité 12000 lignes de code pour modéliser les définitions, les lemmes et leurs preuves, associés aux traces de programmes. Les preuves constituent environ 70 % du travail.

Le développement est disponible à l'adresse suivante :

`https://traclifo.univ-orleans.fr/PaPDAS/wiki/TransactionsInCoq`

Il fonctionne avec la version 8.4 de Coq.

Les définitions et les preuves présentées dans ce chapitre utilisent le formalisme mathématique classique, mais ne sont pas formelles. Une preuve mathématique classique a pour destinataire le lecteur, et de ce fait certaines étapes de routines, ou des résultats classiques supposés connus, sont omis. Une preuve formelle quant à elle, n'omet aucune étape, tous les résultats intermédiaires doivent être prouvés. L'écriture de ces preuves et donc plus long est difficile. Néanmoins la confiance dans ces preuves est accrue car elles sont ensuite vérifiées par un assistant de preuve. L'assistant choisi dans ce travail est Coq [82, 9, 20].

La question du style d'écriture des preuves et des définitions se pose. Coq étant un langage riche, il existe toujours plusieurs manières de modéliser un concept ou de rédiger une preuve. Nous sommes toujours à la recherche de l'équilibre entre l'élégance et la simplicité.

Comme tout développement logiciel, une organisation particulière et des choix de génie logiciel ont été appliqués. Nous détaillons ici l'organisation que le lecteur désireux d'explorer les sources Coq puisse le faire le plus aisément possible. Les différentes notions développées ici ont été divisées en répertoires, fichiers et modules. Un répertoire existe pour chaque grande partie du développement (traces, sémantique opérationnelle, compilation) plus un répertoire `Common` contenant les notions partagées par ces différentes parties.

Le découpage en fichiers suit les règles suivantes :

— Les définitions sont regroupées dans un fichier portant le nom de la notion abordée. Aucun lemme n'est présent. Par exemple les traces ainsi que les prédicats sur celles-ci (*range*, *tribe* ...) sont définis dans `Trace.v`.

— Les lemmes auxiliaires ou de moindre importances sont placés dans un fichier portant le nom de la notion abordée avec le suffixe `_Basics`. Quand trop de lemmes sont présents, ce fichier peut être re-découpé et l'on y ajoute le suffixe sur le prédicat concerné. Par exemple le fichier `Trace_Basics_tribe.v` contient les lemmes auxiliaires portant sur les tribus d'une trace. Lorsque qu'un de ces redécoupages a lieu, un fichier regroupe tous ces lemmes grâce à une suite d'importations.

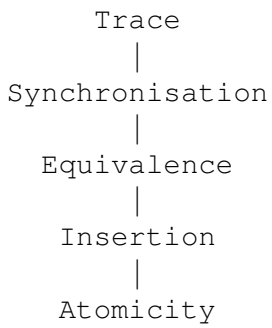


FIGURE 4.6 – *Organisation générale de la formalisation en Coq*

— Enfin les lemmes et théorèmes que l'on souhaite exposer sont présents dans un fichier portant le nom de la notion et le suffixe `_Theory`. Par exemple le lemme 4.1 est présent dans le fichier `Trace_Theory.v`.

Il est à noter que le premier fichier peut être absent car les définitions peuvent être communes, mais pas les résultats. C'est le cas par exemple pour `Insertion.v`. Par la suite lorsque j'évoquerai une notion et son emplacement je ne ferai pas nécessairement référence au fichier avec son suffixe précis.

Les fichiers modélisant les notions vues dans cette section se trouvent dans le répertoire `Transactions`. La contribution principale de ce chapitre est la définition de l'atomicité et de la bonne synchronisation. Comme vu dans le schéma de la preuve du théorème 4.1, les traces équivalentes s'obtiennent en décalant un élément dans la trace. Ces deux notions complémentaires, l'insertion et l'équivalence, sont définies respectivement dans `Insertion` et `Equivalence`. L'équivalence (cf définition 4.4) se base sur les notions de synchronisation, qui sont définies dans `Synchronisation`.

Tout ces notions sont exprimées sur les traces qui sont elles-mêmes définies dans `Trace`.

Ces fichiers et ces dépendances peuvent se résumer à l'organisation présentée dans la figure 4.6.

4.4.2 Choix de formalisation

Des modules sont utilisés afin de gagner en abstraction. Ils sont paramétrés par les noms de section et les adresses. Ainsi nous sommes indépendant de ces définitions mais nous exigeons que ce soit des types dont l'égalité est décidable. Nous avons fait ce choix, mais nous aurions pu utiliser l'axiome du tiers exclu. Nous avons voulu éviter les axiomes pour gagner en modularité. En effet bien que l'utilisation du tiers exclu aurait été tout à fait acceptable, sa combinaison avec d'autres axiomes aurait pu entraîner un certain nombre de problèmes.

Reprenons le paragraphe de la section 4.1.

> Nous supposons des ensembles dénombrables d'emplacement mémoire, de noms de processus léger, et de noms de section, des éléments

notés respectivement ℓ, t et $\mathfrak{s}$, éventuellement avec indice. L'ensemble de valeurs, dont les éléments sont notés v, éventuellement avec indice, contient au moins les emplacements mémoires, les entiers et les noms de processus léger.

Chacun des modules est paramétré par les adresses et les noms de sections. Nous avons précisé que ce sont des ensembles dénombrables, dont l'égalité est décidable. C'est pourquoi nous définissons le type infite, pour préciser que l'on a un ensemble dénombrable. Il est défini par une bijection de $\mathbb{N}$, de la manière suivante:

```
Definition infinite (A: Type) : Type :=
  { bij: (A→nat)∗(nat→A) | ∀(n:nat), (fst bij)((snd bij) n) = n }.
```

Nous avons donc une plage infinie d'adresses et de noms de section, nous pouvons donc toujours trouver un nom de section frais, ou une adresse non allouée. Comme ils sont en paramètres nous pouvons changer de définitions tant qu'elles respectent la spécification.

Dans la formalisation en Coq, les types de valeurs possible sont encodés par un inductif Coq. Une fonction permet ensuite d'obtenir le type Coq effectif modélisant les valeurs de ce type dans les actions. Les types et les valeurs sont détaillés dans la section 5.4.2

4.4.3 Un exemple de preuve

Nous avons prouvé le Théorème 4.1 de manière informelle plus haut. Examinons maintenant la même preuve, mais faite en Coq pour voir les points communs et les différences.

```
Theorem wsync_atomic :
  ∀s,
    wellFormed s →
    wellSynchronized s →
    exists s', compatible s s' ∧ atomic s'.
```

L'énoncé du théorème est très proche de l'original, nous n'avons pas défini explicitement la notion de serialisabilité : c'est une conjonction de compatible (équivalence de traces) et atomic (sériel). La preuve est faite par induction. Comme les traces sont des listes, et que le principe d'induction par défaut est celui qui correspond aux constructeurs, nous devons explicitement déclarer que nous raisonnerons par induction en ajoutant un élément à la fin de la liste : c'est le principe d'induction tr_ind. Le premier cas, le cas de la trace vide, peut être repéré grâce à l'utilisation des *bullets*, des symboles ($-$,+,$*$) permettant de structurer la preuve, en cas et sous-cas. La preuve de la trace vide est relativement immédiate.

```
Proof.
  induction s using tr_ind; intros h_wf h_ws.
  − (exists nil); split; [apply compatible_refl | apply nil_atomic].
```

Dans le cas inductif, la trace a été décomposée en $s \cdot e$. Ici, nous voyons une autre manière de structurer des preuves. La tactique assert permet d'énoncer une propriété et ensuite nous devons la prouver pour pouvoir utiliser ce résultat plus tard dans la preuve.

```
– assert (exists s', compatible s s' ∧ atomic s') as [s' [Ha Hb]].
  {
    assert (wellFormed s) by now apply wellFormed_se_s with e.
    assert (wellSynchronized s).
    apply wellSynchronized_se_s with e; try now inversion h_wf.
    destruct IHs as [s' [Ha Hb]]; eauto.
  }
```

Le but ici est d'utiliser l'hypothèse d'induction IHs, et pour ce faire nous avons besoin de prouver ses prémisses. Quand la preuve pour obtenir un assert est trop longue et que la propriété est en quelque sorte une propriété basique de notre concept, nous énonçons plutôt un lemme et le prouvons séparément pour améliorer la lisibilité de la preuve `Coq`.

```
assert (s · e ∼= s' · e) by now apply compatible_S.
```

Le lemme compatible_s est résumé dans le deuxième paragraphe de la preuve informelle.

```
assert (wellFormed (s' · e))
        by now apply compatible_wellFormed with (s · e).
assert (wellSynchronized (s' · e))
        by now apply compatible_wellSynchronized with (s · e).
assert (wellFormed s) by now apply wellFormed_se_s with e.
assert (wf_occurences (s · e)) by (inversion h_wf;auto).
```

Ensuite nous devons raisonner s'il existe ou non une section pendante la plus à gauche, telle que le processus léger de l'évènement e n'est pas dans la tribu de cette section. Les deux cas sont les seules possibilités : nous avons montré que le prédicat exclude est décidable, et destruct nous permet de considérer les deux alternatives séparément :

```
destruct e as [t a];
  destruct (exclude_dec s' (wellFormed_se_s s' (t,a) H0) t)
  as [h_noExclude | h_outerExclude].
```

Dans le premier cas, il n'y a pas de telle section :

```
+ assert (atomic (s' · (t,a))).
    {
      assert (∼ conflicts s' t) by
          (intro h_conflicts; destruct h_conflicts as [p h_p];
          now elim (h_noExclude p)).
      now apply noExclude_atomic.
```

```
    }
    (exists (s' · (t,a))); tauto.
```

L'autre cas requiert plus de travail. Premièrement nous avons besoin d'une assertion pour exhiber la position i où la section s'ouvre :

```
+ destruct h_outerExclude as [p Hp].
    assert (exists i, action_of (pi i s') == Open p)
      as [i Hi] by now apply outerExclude_open with t.
```

Ensuite nous construisons la trace $s'' = s' \upharpoonleft_i (t, a)$:

```
    assert (exists s'', insertion i s' (t,a) == s'') as [s'' Hs'' ].
    {
      assert (i ≤length s')
        as Hv
          by (assert (i < length s') by eauto with nth_error; intuition).
      destruct (insertion_defined i s' (t,a) Hv) as [s'' Hs''].
      exists s''; assumption.
    }
```

Ensuite nous prouvons l'équivalence de s' et de la trace initiale :

```
    assert (∀ k : threadId,
                      i ≤k < length s' →
                      ∼synchronizeWith (s' · (t, a)) k (length s'))
            by (inversion Hp; now apply exclude_noSync with p).
    assert (s · (t,a) ∼= s'').
    {
      apply compatible_trans with (s' · (t,a)).
      − apply compatible_S;auto.
      − assert (∀ k : threadId,
                      i ≤k < length s' →
                      ∼synchronizeWith (s' · (t, a)) k (length s'))
            by (inversion Hp; now apply exclude_noSync with p).
        assert (exclude s' p t) by now inversion Hp.
        now apply insertion_compatible with i p.
    }
```

en utilisant la transitivité de l'équivalence (compatible_trans) et de ins_{eq}. Pour terminer la preuve, nous avons juste besoin de prouver que la section est atomique.

```
    assert (atomic s'').
    now apply outerExclude_insertion_atomic with s' p t a i.
    exists s''; tauto.
Qed.
```

Cette preuve est assez lisible, car c'est grâce au soin que nous avons porté au style en utilisant des assertions et en déportant certains détails fastidieux dans les lemmes. Nous essayons d'appliquer systématiquement ce style pour les preuves et pour les résultats principaux. Par comparaison, certaines preuves de certains lemmes basiques ne sont pas si bien structurées.

4.4.4 Correspondance

Les tableaux suivants 4.7 et 4.8 donne la correspondance entre les définitions et les lemmes et leur équivalent en Coq. Pour chacun d'entre eux nous indiquons son nom en Coq et le fichier Coq. Tous les fichiers sont dans le répertoire `Transactions` de l'archive.

Nom	**Coq**	**Fichier**
section name	t (in module Permission)	`Permission.v`
actions	action	`Trace.v`
events	t (in module Event)	`Trace.v`
traces	Tr	`GenericTrace.v`
$\Subset$	sec_order	`Trace.v`
$\frown$	#	`Trace.v`
owns	owns	`Trace.v`
father	father	`Trace.v`
$\prec$	precedes	`Trace.v`
see	see	`Trace.v`
range	range	`Trace.v`
tribeChildren	tribeChildren	`Trace.v`
tribe	tribe	`Trace.v`
excludes	exclude	`Trace.v`
insertion	insertion	`Insertion.v`
Definition 4.1	wellSynchronized	`Synchronisation.v`
Definition 4.2	atomic	`Atomicity.v`
Definition 4.3	atomic and compatible	
Definition 4.4	compatible	`Equivalence.v`
$\mathbf{wf}_1$	wf_occurences	`Trace.v`
$\mathbf{wf}_2$	wf_open_close	`Trace.v`
$\mathbf{wf}_3$	wf_seq_order	`Trace.v`
$\mathbf{wf}_4$	wf_fork	`Trace.v`
$\mathbf{wf}_5$	wf_join	`Trace.v`
$\mathbf{wf}_6$	wf_join_all_closed	`Trace.v`
$\mathbf{wf}_7$	wf_join_see_fork	`Trace.v`
$\mathbf{wf}_8$	wf_mutual_exclusion	`Trace.v`

FIGURE 4.7 – *Table de correspondance avec les définitions Coq*

Nom	**Coq**	**Fichier**
Lemma 4.1	wellFormed_prec_tribe	`Trace_Theory.v`
Lemma 4.2	see_synchronizeWith	`Synchronisation.v`
Proposition 4.1	sw_in_tribe	`Synchronisation.v`
Proposition 4.2	compatible_wellSynchronised compatible_wellFormed	`EquivalenceTheory.v`
$\mathbf{ins}_{\mathbf{eq}}$	insertion_compatible	`SyncInsertion.v`
$\mathbf{ins}_{\mathbf{tribe}}$	insertion_tribe	`SyncInsertion.v`
Theorem 4.1	wsync_atomic	`AtomicityFinal.v`

FIGURE 4.8 – *Table de correspondance avec les énoncés et preuves Coq*

AFJ: un langage à transactions emboîtées et processus légers

5

Sommaire

Une version de ce chapitre est paru dans : Frédéric Dabrowski, Frédéric Loulergue, Thomas Pinsard, Nested Atomic Semantics with Thread Escape: An Operational Semantics, *International Conference on Parallel and Distributed Computing, Applications and Technologies (PDCAT), pages 29-36, IEEE Computer Society, Décembre 2013.*

Nous venons de présenter un ensemble de conditions qui peuvent être vues comme une spécification pour la sémantique de langages de programmation impératifs avec sections atomiques emboîtées et parallélisme interne aux sections. L'étape suivante est de proposer un langage qui vérifie ces conditions. Nous avons appelé ce langage *Atomic Fork Join* (AFJ) que nous détaillons dans un premier temps à travers sa syntaxe et quelques exemples. Nous proposons ensuite une sémantique opérationnelle et nous démontrons que celle-ci vérifie les conditions de bonnes formations détaillées au chapitre précédent. Nous concluons par la présentation de la réalisation partielle des éléments de ce chapitre en Coq.

5.1 Le Language *Atomic Fork Join*

Nous supposons des ensembles dénombrables disjoints d'emplacements mémoire ($\mathbb{L}$), de noms de processus léger ($\mathbb{T}$), dont les éléments sont notés respectivement ℓ et t,

éventuellement avec des indices. Il y a également un emplacement mémoire particulier noté `null` qui ne peut être alloué. L'ensemble des valeurs, dont les éléments sont notés v, éventuellement avec des indices, contient au moins les emplacements mémoires, les entiers, les booléens ($\mathbb{B}$) et les noms de processus légers.

$$v \quad ::= \quad n \mid \ell \mid t \mid b \text{ où } n \in \mathbb{Z}, \ell \in \mathbb{L}, t \in \mathbb{T} \text{ et } b \in \mathbb{B}$$

Dans les grammaires suivantes, $\mathcal{X}$ signifie l'ensemble dénombrable des variables locales, $\bar{e}$ est un tuple d'expressions et `op` indique une opération prédéfinie sur les valeurs, d est l'ensemble des valeurs que les utilisateurs peuvent écrire en tant que constantes, par exemple 42 pour une constante entière, ou `true` pour une constante booléenne. Cet ensemble exclut les identifiants de processus léger.

$$\begin{array}{lll} d & ::= & n \mid b \mid \ell \\ e & ::= & d \mid x \mid \texttt{op}(\bar{e}) \text{ où } x \in \mathcal{X} \end{array}$$

L'ensemble des opérations prédéfinies n'est pas détaillé ici, mais contient les opérations arithmétiques et booléennes usuelles. Un programme AFJ est un ensemble de méthodes et une instruction principale. Une méthode est définie par son nom, un argument et un corps qui est une séquence d'instructions. La grammaire pour les programmes, méthodes et instructions est la suivante :

$$\begin{array}{rcl} mth & ::= & \overline{m(x)\ c} \\ r & ::= & \overline{mth}\ c \\ c & ::= & \mid \texttt{skip} \mid c;c \mid \texttt{if}\ b\ \texttt{then}\ c\ \texttt{else}\ c \\ & & \mid \texttt{while}\ b\ \texttt{do}\ c \mid x := e \mid x := y[e'] \mid x[e'] := e'' \\ & & \mid x := \texttt{allocate}(e) \mid \texttt{dispose}(e) \\ & & \mid \texttt{atomic}\ \{c\} \mid x = \texttt{fork}(m, e) \mid \texttt{join}\ e \end{array}$$

Le langage AFJ a une instruction `skip` et les structures de contrôle séquentielles habituelles : séquence, conditionnel et boucle. Une variable locale peut être affectée avec la valeur d'une expression, ou la valeur contenue à un emplacement mémoire donné. On peut lire ou écrire dans un emplacement mémoire. Cependant pour accéder à un tel emplacement mémoire, il doit avoir été précédemment alloué grâce à l'instruction `allocate` qui prend en argument le nombre de cellule mémoire à allouer de façon contiguë et retourne l'adresse de la première cellule mémoire allouée. La désallocation de mémoire précédemment allouée est faite par la primitive `dispose` qui libère un emplacement mémoire donné.

La primitive `atomic` permet d'indiquer que la séquence d'instructions donnée doit être exécutée de manière atomique. La primitive `fork` permet la lancer l'exécution, par un processus léger nouvellement créé, d'une méthode avec un argument donné. La primitive `fork` retourne l'identifiant du processus léger créé. La primitive `join` permet de synchroniser le processus léger courant avec le processus léger dont l'identifiant est donnée en paramètre. Un très court exemple est proposé dans la Figure 5.1.

```
m(x)
  atomic{
    x[0] := x[0]+1;
  }

x := allocate(1);
```

```
x[0] := 1;
u := fork(m,x);
atomic{
  x[0] := x[0]+1;
};
join(u);
```

FIGURE 5.1 – *Exemple de programme AFJ*

5.2 UNE SÉMANTIQUE OPÉRATIONNELLE POUR *Atomic Fork Join*

Nous décrivons maintenant une sémantique opérationnelle qui satisfait la spécification énoncée dans le chapitre précédent (Chapitre 4).

5.2.1 États, actions et évènements

Chaque processus léger a son propre environnement local. Les processus légers se partagent le tas. L'environnement local est une fonction partielle des variables vers les valeurs. Le tas est une fonction partielle prenant un emplacement mémoire et un décalage et retournant une valeur. La valeur d'une cellule mémoire non-allouée est indéfinie. L'ensemble des processus légers est une fonction partielle des identifiants de processus légers vers une paire constituée du code restant à exécuter par le processus léger, et son environnement local. Dans la suite, $\rightharpoonup$ dénote une fonction partielle. $\mathcal{V}$ et $\mathcal{C}$ signifie respectivement l'ensemble des valeurs et instructions.

$$\begin{array}{lcll} \rho & \in & \mathcal{E} = \mathcal{X} \rightharpoonup \mathcal{V} & \text{environnent local} \\ \sigma & \in & \mathbb{L} \rightharpoonup \mathbb{N} \rightharpoonup \mathcal{V} & \text{tas} \\ \phi & \in & \mathbb{T} \rightharpoonup \mathcal{C} \times \mathcal{E} & \text{ensemble de processus légers} \end{array}$$

Nous utilisons la notation tableau pour manipuler le tas. Pour obtenir une valeur du tas, nous utilisons $\sigma(\ell)[n]$ qui signifie $\sigma(\ell, n)$. Pour modifier le tas nous utilisons la notation $\sigma \cdot [\ell + n \mapsto v]$ définie par :

$$\begin{cases} \sigma \cdot [\ell + n \mapsto v](\ell')[n'] & = & \sigma(\ell')[n'] \text{ si } \ell' \neq \ell, n' \neq n \\ \sigma \cdot [\ell + n \mapsto v](\ell)[n] & = & v \end{cases}$$

Nous utilisons également la variante $\sigma \cdot [\ell \mapsto [\ldots]^n]$ où $[\ldots]^n$ est une expression représentant un « tableau » de n valeurs.

L'évaluation des expressions, notée $[\![e]\!]$, est définie classiquement, comme par exemple dans [67]. Les contextes de réduction locaux, notés C, permet de rendre explicite quelle instruction est évaluée par un processus léger. Ils sont définis comme suit :

$$C ::= \square \mid C; c$$

Comme il est classique, $C[c]$ dénote l'instruction obtenue en substituant l'instruction c au symbole $\square$ dans le contexte C.

Pour assurer l'atomicité dans la sémantique opérationnelle, nous utilisons une structure récursive, appelée bulle et notée B. Un programme démarre toujours dans un état initial et son instruction principale est exécutée par le processus léger initial t_e dans le programme. La définition formelle d'une bulle est :

$$\mathrm{B} \quad ::= \quad (\!|\, \phi; \mathrm{B}^\circ \,|\!)^{\mathfrak{s},t}_{C} \qquad\qquad \mathrm{B}^\circ \quad ::= \quad \circ \mid \mathrm{B}$$

où $\mathfrak{s}$ est le nom de la section, t est le processus léger qui démarre la section, c'est-à-dire le propriétaire de la section, C est le contexte local, permettant de reprendre l'exécution après la fin de la section. Il est a noter que le programme débute dans une bulle initiale, dont le nom est $\mathfrak{s}_e$.

Par exemple, considérons un programme contenant l'instruction :

$$C_0[\texttt{atomic}\ \{C_1[\texttt{atomic}\ \{c\}]\}]$$

Une exécution de ce programme peut alors atteindre un état où la bulle est :

$$(\!|\, \phi; (\!|\, (c,\rho)^{t_1}; \circ \,|\!)^{\mathfrak{s}_1,t_1}_{C_1} \,|\!)^{\mathfrak{s}_e,t_e}_{C_0}$$

juste après que le programme commence à exécuter l'instruction $\texttt{atomic}\ \{c\}$. Si dans C_1, nous avons la création d'un nouveau processus léger qui ouvré également une section atomique, elle ne peut être ouverte que quand la bulle $\mathfrak{s}_1$ est fermée. Il est uniquement possible d'avoir une des sections concurrentes ouvertes à la fois. Intuitivement, cela assure l'atomicité.

Dans la structure de bulle, il y a souvent plusieurs processus légers possibles à exécuter, éventuellement à des sous-bulles différentes. Ainsi pour exprimer la sémantique, nous avons besoin de la notion de contexte de bulle. Un contexte de bulle est noté K et est défini de la manière suivante :

$$K ::= [] \mid (\!|\, \phi; K \,|\!)^{\mathfrak{s},t}_{C}$$

Nous utilisons la notation $t \in B$ si $B = (\!|\, \phi; B_0 \,|\!)'$ et soit $\phi(t)$ est défini, soit $t \in B_0$. Par abus de notation, nous l'utilisons aussi pour les contextes de bulle et B°. Nous l'utilisons également pour l'appartenance dans un ensemble de processus léger, $t \in \phi$ si $\phi(t)$ est défini. Nous disons que $\mathfrak{s} \in B$ si $B = (\!|\, \phi; B_0 \,|\!)^{\mathfrak{s}_1}$ et soit $\mathfrak{s} = \mathfrak{s}_1$, soit $\mathfrak{s} \in B_0$ (noté ici que nous n'avons pas spécifié le processus léger propriétaire de la bulle, par abus de notation car il n'intervient pas dans la définition).

5.2.2 Règles

La sémantique opérationnelle est définie par un ensemble de relations. La première est une relation sur les états de programme consistant en une bulle, un tas et un ensemble de noms de section utilisés :

$$\vdash_p \ B, \sigma, S \ \xrightarrow{a}_t \ B', \sigma', S'$$

Elle est définie dans la Figure 5.4 par la règle (**main**) et sélectionne simplement la sous-bulle sur laquelle agir, c'est-à-dire le contexte de bulle. L'exécution réelle est formalisée par la prochaine relation. Les autres règles sur les sections atomiques ont la forme :

$$\mathrm{K} \vdash_p \ B, \sigma, S \ \xrightarrow{a}_t \ B', \sigma', S'$$

et sont définies dans la Figure 5.4. Elles sont utilisées pour ouvrir/fermer une section ou pour exécuter des instructions sans rapport avec les sections et utilisant une relation qui définit le comportement inter-processus léger. Cette relation (Figure 5.3) a la forme :

$$K, B^\circ, t_0 \vdash_p \ \phi, \sigma \ \xrightarrow{a}_t \ \phi', \sigma'$$

où t_0 (resp. B°) est le propriétaire de la section (resp. la sous-bulle directe de la bulle) dans lequel le processus léger t s'exécute. Il est important d'avoir cette information pour générer des noms de processus léger frais ou pour vérifier la terminaison de processus légers.

La dernière relation, utilisée dans la précédente, traite seulement les instructions « séquentielles ». Elle est défini dans la Figure 5.2 et a la forme :

$$c, \rho, \sigma \ \xrightarrow{a} \ c', \rho', \sigma'$$

La règle (**open**) décrit la création de sous-sections. Premièrement, il ne doit déjà pas y avoir de sous-sections. Dans la bulle représentant la section courante, une sous-bulle est ajoutée. Le nom de cette nouvelle bulle doit être unique. Cela est assuré par le prédicat *fresh* qui utilise l'ensemble des noms déjà pris. Ce nouveau nom est ensuite ajouté à cet ensemble. Le contexte local est sauvegardé pour reprendre l'exécution du processus léger dans le futur. La nouvelle sous-bulle contient seulement un processus léger, qui est le processus léger propriétaire et ses instructions sont celles protégées par la section.

La règle (**close**) décrit la fermeture d'une section : elle peut être fermée quand le processus léger propriétaire est terminée, c'est-à-dire associé à la commande `skip`. Nous pouvons noter ici qu'il peut y avoir d'autres processus légers continuant à s'exécuter à ce niveau. Cet ensemble de processus légers est ajouté à l'ensemble de la section parente. Le processus léger propriétaire de la section fermante peut continuer son exécution grâce au contexte local sauvegardé. La dernière réduction (**inter**) modifie seulement l'ensemble des processus légers et le tas.

La règle (**fork**) crée un nouveau processus léger. Le prédicat *fresh* assure que le nouveau identifiant de processus léger n'est pas dans l'état courant et, comme les noms de processus léger restent dans l'état, n'a jamais été utilisé. Le nouveau processus léger est associé au corps de la méthode donnée en paramètre dans l'instruction `fork`, et son environnement local est défini pour le paramètre formel de la méthode et a comme valeur son paramètre effectif.

La règle (**join**) permet d'attendre un processus léger terminé. Le prédicat *term*, assure que le processus léger est terminé, c'est-à-dire que l'instruction associée doit être `skip` et ne doit être propriétaire d'aucune section. Ce prédicat prend en paramètre

un contexte de bulle, une bulle, et l'ensemble des processus légers de ce niveau afin de retrouver le processus léger.

La règle (**intra**) fait un pas de réduction pour un processus léger et est décrit dans la Figure 5.2 avec des instructions plus classiques (conditionnel, boucle, ...). Notez que dans la règle (**alloc**), les nouvelles cellules ont la valeur par défaut 0, et nous vérifions que ℓ est libre dans le tas grâce à la condition $\ell \notin dom(\sigma)$.

Chaque règle de réduction est étiquetée avec une action (décrit dans le Chapitre 4) et un processus léger. Ces deux éléments permettent de créer un évènement, et plusieurs pas de réduction d'un programme p partant de l'état initial $\Sigma_\epsilon(p)$ créent une trace. L'état initial est composé de la bulle initiale avec le nom de section initiale $\mathfrak{s}_\epsilon$ et le nom du processus léger initial t_ϵ. Le processus léger t_ϵ est associé à la commande principale c du programme. Le tas initial ne contient rien. L'ensemble initial des noms de section utilisés contient seulement le nom de section initiale $\mathfrak{s}_\epsilon$.

Un exemple d'une séquence de réductions du programme présenté dans la Figure 5.1 est détaillé dans la Figure 5.6. Celui-ci illustre un certain nombre de caractéristiques de notre langage (création et synchronisation de processus léger, sections atomiques). La même cellule mémoire ($x[0]$) est accédée en écriture et en lecture par deux processus légers. Ces accès sont bien synchronisés par l'emploi de section atomique. En effet nous pouvons voir sur l'exemple qu'il n'y a pas d'entrelacement d'exécution des deux sections atomiques. Celles-ci sont prêtes à démarrer au même moment. Le choix de débuter par la section du processus léger t_1 est arbitraire. Une fois débuté la sous-bulle créée interdit à la section concurrente de démarrer, garantissant ainsi l'exclusion mutuelle. Nous pouvons voir la figure 5.5 la structure de bulle au moment du trait vertical.

5.3 Propriétés de la sémantique

Les propriétés de bonne formation sur les traces définies dans le chapitre 4 peuvent être vues en tant que spécification pour les sémantique opérationnelles et les implantations de langages impératifs avec emboîtement de sections atomiques et échappement de processus légers. Nous voulons montrer que cette spécification est satisfaite par la sémantique opérationnelle que nous proposons. Notre principal théorème énonce que chaque trace produite par un programme vérifie les conditions de bonne formation. Commençons avec quelques définitions.

Nous disons qu'une bulle B est *atteignable* par un programme p démarrant de l'état initial $\Sigma_\epsilon(p)$ et générant une trace s, s'il y a une séquence de réduction où la séquence d'évènements est égal à s, démarrant de $\Sigma_\epsilon(p)$, qui termine dans un état où B est le premier composant. Cela est noté ${}^pB_s$. Nous définissons la bulle atteignable en i comme la bulle atteinte après le i^{me} évènement de la trace s produite par le programme r partant de l'état $\Sigma_\epsilon(p)$. Cela est noté ${}^pB^i_s$. Si le contexte est clair, nous ne spécifierons pas le programme, et quelque fois pas la trace mais uniquement la position et nous l'écrivons B^i.

$$(x := e,\ \rho),\ \sigma \xrightarrow{\tau} (\texttt{skip},\ \rho[x \mapsto v]),\ \sigma \qquad \text{si } [\![e]\!] = v \quad \textbf{(assign)}$$

$$(x := \texttt{allocate}(e),\ \rho),\ \sigma \xrightarrow{\texttt{alloc}\ \ell\ n} (\texttt{skip},\ \rho[x \mapsto \ell]),\ \sigma \cdot [\ell \mapsto [0]^n] \quad \textbf{(alloc)}$$
$$\text{si } [\![e]\!] = n, n > 0, \ell \neq \texttt{null}, \text{ et } \ell \notin dom(\sigma)$$

$$(\texttt{dispose}(e),\ \rho),\ \sigma \xrightarrow{\texttt{free}\ \ell} (\texttt{skip},\ \rho),\ \sigma_{|dom(\sigma)-\{\ell\}} \quad \textbf{(dispose)}$$
$$\text{si } [\![e]\!] = \ell \text{ et } \ell \in dom(\sigma)$$

$$(x := y[e],\ \rho),\ \sigma \xrightarrow{\texttt{read}\ \ell\ n\ v} (\texttt{skip},\ \rho[x \mapsto v]),\ \sigma \quad \textbf{(get)}$$
$$\text{si } \rho(y) = \ell,\ [\![e]\!] = n,\ \sigma(\ell)[n] = v$$

$$(x[e_1] := e_2,\ \rho),\ \sigma \xrightarrow{\texttt{write}\ \ell\ n\ v} (\texttt{skip},\ \rho),\ \sigma \cdot [\ell + n \mapsto v] \quad \textbf{(put)}$$
$$\text{si } \rho(x) = \ell,\ [\![e_1]\!] = n, [\![e_2]\!] = v, \ell \in dom(\sigma) \text{ et } \sigma(l)[n] \text{ défini}$$

$$(\texttt{skip}; c,\ \rho),\ \sigma \xrightarrow{\tau} (c,\ \rho),\ \sigma \quad \textbf{(sequence)}$$

$$(\texttt{while}\ b\ \texttt{do}\ c,\ \rho),\ \sigma \xrightarrow{\tau} (c; \texttt{while}\ b\ \texttt{do}\ c,\ \rho),\ \sigma \qquad \text{si } [\![b]\!] = \texttt{true} \quad \textbf{(loop}_\textbf{t}\textbf{)}$$

$$(\texttt{while}\ b\ \texttt{do}\ c,\ \rho),\ \sigma \xrightarrow{\tau} (\texttt{skip},\ \rho),\ \sigma \qquad \text{si } [\![b]\!] = \texttt{false} \quad \textbf{(loop}_\textbf{f}\textbf{)}$$

$$(\texttt{if}\ b\ \texttt{then}\ c_1\ \texttt{else}\ c_2,\ \rho),\ \sigma \xrightarrow{\tau} (c_1,\ \rho),\ \sigma \qquad \text{si } [\![b]\!] = \texttt{true} \quad \textbf{(cond}_\textbf{t}\textbf{)}$$

$$(\texttt{if}\ b\ \texttt{then}\ c_1\ \texttt{else}\ c_2,\ \rho),\ \sigma \xrightarrow{\tau} (c_2,\ \rho),\ \sigma \qquad \text{si } [\![b]\!] = \texttt{false} \quad \textbf{(cond}_\textbf{f}\textbf{)}$$

FIGURE 5.2 – *Sémantique opérationnelle de AFJ : règles intra-processus léger*

$$\mathrm{K}, \mathrm{B}^\circ, t_0 \vdash_p \phi \cdot (\mathrm{C}[c], \rho)^t, \sigma \xrightarrow{a}_t \phi \cdot (\mathrm{C}[c'], \rho')^t, \sigma' \qquad \textbf{(intra)}$$
$$\text{si } (c,\ \rho),\ \sigma \xrightarrow{a} (c',\ \rho'),\ \sigma'$$

$$\mathrm{K}, \mathrm{B}^\circ, t_0 \vdash_p \phi \cdot (\mathrm{C}[y{:=}\texttt{fork}(m,e)], \rho)^{t_1}, \sigma \xrightarrow{\texttt{fork } t_2}_{t_1} \phi \cdot (\mathrm{C}[\texttt{skip}], \rho[y \mapsto t_2])^{t_1} \cdot (c, [x \mapsto v])^{t_2}, \sigma \qquad \textbf{(fork)}$$
$$\text{si } [\![e]\!] = v,\ m(x)\ c \text{ méthode de } p, \text{ et } \mathit{fresh}_{\mathrm{K},\phi\cdot(\mathrm{C}[\texttt{fork}(m,e)],\rho)^{t_1},\mathrm{B}^\circ,t_0}(t_2)$$

$$\mathrm{K}, \mathrm{B}^\circ, t_0 \vdash_p \phi \cdot (\mathrm{C}[\texttt{join } e], \rho)^{t_1}, \sigma \xrightarrow{\texttt{join } t_2}_{t_1} \phi \cdot (\mathrm{C}[\texttt{skip}], \rho)^{t_1}, \sigma \qquad \textbf{(join)}$$
$$\text{si } [\![e]\!] = t_2 \text{ et } \mathit{term}_{\mathrm{K},\phi,\mathrm{B}^\circ,t_0}(\mathrm{t}_2)$$

FIGURE 5.3 – *Sémantique opérationnelle de AFJ : règles inter-processus légers*

$$\mathrm{K} \vdash_p (\!|\ \phi; \mathrm{B}^\circ\ |\!)^{\mathfrak{s}_0,t_0}_{\mathrm{C}_0}, \sigma, S \xrightarrow{a}_t (\!|\ \phi'; \mathrm{B}^\circ\ |\!)^{\mathfrak{s}_0,t_0}_{\mathrm{C}_0}, \sigma', S \qquad \textbf{(inter)}$$
$$\text{si } \mathrm{K}, \mathrm{B}^\circ, t_0 \vdash_r \phi, \sigma \xrightarrow{a}_t \phi', \sigma'$$

$$\mathrm{K} \vdash_p (\!|\ \phi \cdot (\mathrm{C}[\texttt{atomic } \{c\}], \rho)^t; \circ\ |\!)^{\mathfrak{s}_0,t_0}_{\mathrm{C}_0}, \sigma, S \xrightarrow{\texttt{open } \mathfrak{s}_1}_t (\!|\ \phi; (\!|\ (c, \rho)^t; \circ\ |\!)^{\mathfrak{s}_1,t}_{\mathrm{C}}\ |\!)^{\mathfrak{s}_0,t_0}_{\mathrm{C}_0}, \sigma, S \cup \{\mathfrak{s}_1\} \qquad \textbf{(open)}$$
$$\text{si } \mathit{fresh}_S(\mathfrak{s}_1)$$

$$\mathrm{K} \vdash_p (\!|\ \phi_0; (\!|\ \phi_1 \cdot (\texttt{skip}, \rho)^t; \circ\ |\!)^{\mathfrak{s}_1,t}_{\mathrm{C}_1}\ |\!)^{\mathfrak{s}_0,t_0}_{\mathrm{C}_0}, \sigma, S \xrightarrow{\texttt{close } \mathfrak{s}_1}_t (\!|\ \phi_0 \cdot \phi_1 \cdot (\mathrm{C}_1[\texttt{skip}], \rho)^t; \circ\ |\!)^{\mathfrak{s}_0,t_0}_{\mathrm{C}_0}, \sigma, S \qquad \textbf{(close)}$$

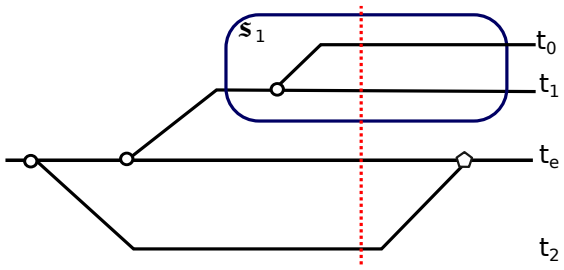
$$\vdash_p K[B], \sigma, S \xrightarrow{a}_t K[B'], \sigma', S' \qquad \textbf{(main)}$$
$$\text{si } K \vdash_p B, \sigma, S \xrightarrow{a}_t B', \sigma', S'$$

FIGURE 5.4 – *Sémantique opérationnelle de AFJ : sections atomiques*

Nous définissons la *séquence caractéristique* d'une bulle B, noté $cs(B)$, comme la liste de paire (processus léger propriétaire, nom de section) de chaque niveau excepté celui du plus haut niveau. Nous excluons le plus haut niveau car il a quelques spécificités (il n'y a pas de `open` pour la section ni de `fork` pour le processus léger).

$$cs((\!|\ \phi; B_0\ |\!)^{\mathfrak{s}_\epsilon,t_\epsilon}) = \epsilon$$
$$cs((\!|\ \phi; B_0\ |\!)^{\mathfrak{s},t}) = (t, \mathfrak{s}) \cdot cs(B_0) \qquad \text{si } t \neq t_\epsilon \text{ et } \mathfrak{s} \neq \mathfrak{s}_\epsilon$$

Pour la preuve des conditions, nous avons besoin des résultats suivants sur la séquence caractéristique, dont les preuves sont données en section 5.A :



$$(\!| t_e, t_2; (\!| t_0, t_1; \circ |\!)^{\mathfrak{s}_1,t_1}_{C_2} |\!)^{\mathfrak{s}_e,t_e}_{C_1}$$

FIGURE 5.5 – *Exemple de structure de bulles*

Le lemme suivant établit le lien entre deux séquences caractéristiques de deux bulles produites par un même programme pour une même exécution mais à deux point d'exécutions différents.

Lemme 5.1 *Pour tout programme p, trace s, positions i, j, nom de section $\mathfrak{s}$, tels que s est la trace générée par l'exécution du programme p partant de l'état initial et $range_s\ \mathfrak{s}\ (i,j)$, nous avons pour toute position k telle que $i \leq k < j$ si c'est un intervalle fermé (la section $\mathfrak{s}$ est close), ou $i \leq k \leq j$ si c'est un intervalle ouvert (la section $\mathfrak{s}$ est pendante), que la séquence caractéristique $cs({}^pB^i_s)$ est un préfixe de $cs({}^pB^k_s)$.*

Le lemme suivant établit le lien de propriété du processus léger sur la section de chaque paire de la séquence caractéristique.

Lemme 5.2 *Pour tout nom de section $\mathfrak{s}$, processus léger t et bulle B atteignable par un programme p produisant une trace s à partir de l'état initial, tels que $(t,\mathfrak{s}) \in cs(B)$, alors nous avons $owner_s\ \mathfrak{s}\ t$.*

Lemme 5.3 *Pour tout programme p, trace s, bulle B tels que $B = {}^pB_s$, et pour toute séquence l telle que l est un préfixe de $cs(B)$, alors pour tout $(t,\mathfrak{s}) \in l$ et pour tout $(t',\mathfrak{s}') \in cs(B)$ et pas dans l, nous avons $tribe_s\ \mathfrak{s}\ t'$.*

Proposition 5.1 *Toute trace s produite par un programme p partant de l'état initial satisfait la condition ($\mathbf{wf_1}$), i.e. que chaque action* `open` $\mathfrak{s}$, `close` $\mathfrak{s}$ *et* `fork` t *a lieu au plus une fois dans s.*

Démonstration. La preuve est faite par induction sur la trace s. La cas où la trace est vide est trivial. Nous devons prouver la condition sur la trace $s \cdot (t,a)$, avec l'hypothèse d'induction sur la trace s. Si a n'est pas une action `open` $\mathfrak{s}$, `close` $\mathfrak{s}$ ou `fork` t, ou n'apparaît pas dans s, la preuve est immédiate. Nous supposons maintenant que a est égale à `open` $\mathfrak{s}$, `close` $\mathfrak{s}$ ou `fork` t, et apparaît dans la trace s. Nous voulons montrer qu'il y a une contradiction. Soit la règle de réduction $K \vdash B_0, \sigma_0, P_0 \xrightarrow{a}_{t_0} B_1, \sigma_1, S_1$ qui produit cette action et soit la dernière règle de réduction $K' \vdash B'_0, \sigma'_0, S'_0 \xrightarrow{a}_{t} B'_1, \sigma'_1, S'_1$. Nous procédons par cas sur a:

$$\langle\!| (x := \mathit{allocate}(1); x[0] := 1; y := \mathit{fork}(m, x); \texttt{atomic}\ \{x[0] := x[0] + 1\}; \texttt{join}\ y, \emptyset)^t; \circ |\!\rangle^{\mathfrak{s}_1,t}_{\square}, \emptyset, \{\mathfrak{s}_1\}$$
$$\xrightarrow{\texttt{alloc}\ \ell\ 1}_t$$
$$\langle\!| x[0] := x[0] + 1; y := \mathit{fork}(m, x); \texttt{atomic}\ \{x[0] := x[0] + 1\}; \texttt{join}\ y, [x \mapsto \ell])^t; \circ |\!\rangle^{\mathfrak{s}_1,t}_{\square}, [\ell \mapsto 0], \{\mathfrak{s}_1\}$$
$$\xrightarrow{\texttt{write}\ \ell\ 0\ 1}_t$$
$$\langle\!| (y := \mathit{fork}(m, x); \texttt{atomic}\ \{x[0] := x[0] + 1\}; \texttt{join}\ y, [x \mapsto \ell])^t; \circ |\!\rangle^{\mathfrak{s}_1,t}_{\square}, \sigma \cdot [\ell \mapsto [1]], \{\mathfrak{s}_1\}$$
$$\xrightarrow{\texttt{fork}\ t_2}_t$$
$$\langle\!| \begin{array}{l} (\texttt{atomic}\ \{x[0] := x[0] + 1\}; \texttt{join}\ y), [y \mapsto t_2, x \mapsto \ell])^t \cdot \\ (\texttt{atomic}\ \{x[0] := x[0] + 1\}, [x \mapsto \ell])^{t_2} \end{array} ; \circ |\!\rangle^{\mathfrak{s}_1,t}_{\square}, [\ell \mapsto 1], \{\mathfrak{s}_1\}$$
$$\xrightarrow{\texttt{open}\ \mathfrak{s}_2}_t$$
$$\langle\!| \begin{array}{l} (\texttt{atomic}\ \{x[0] := x[0] + 1\}, [x \mapsto \ell])^{t_2} \\ \langle\!| (x[0] := x[0] + 1, [y \mapsto t_2, x \mapsto \ell])^t; \circ |\!\rangle^{\mathfrak{s}_2,t}_{\texttt{join}\ y} \end{array} |\!\rangle^{\mathfrak{s},t}_{\square}, [\ell \mapsto 1], \{\mathfrak{s}_1, \mathfrak{s}_2\}$$
$$\xrightarrow{\texttt{read}\ \ell\ 0\ 1}_t$$
$$\langle\!| (\texttt{atomic}\ \{x[0] := x[0] + 1\}, [x \mapsto \ell])^{t_2}; \langle\!| (x[0] := 1 + 1, [y \mapsto t_2, x \mapsto \ell])^t; \circ |\!\rangle^{\mathfrak{s}_2,t}_{\texttt{join}\ y} |\!\rangle^{\mathfrak{s}_1,t}_{\square}, [\ell \mapsto 1], \{\mathfrak{s}_1, \mathfrak{s}_2\}$$
$$\xrightarrow{\texttt{write}\ \ell\ 0\ 2}_t$$
$$\langle\!| (\texttt{atomic}\ \{x[0] := x[0] + 1\}, [x \mapsto \ell])^{t_2}; \langle\!| (\texttt{skip}, [y \mapsto t_2, x \mapsto \ell])^t; \circ |\!\rangle^{\mathfrak{s}_2,t}_{\texttt{join}\ y} |\!\rangle^{\mathfrak{s}_1,t}_{\square}, [\ell \mapsto 2], \{\mathfrak{s}_1, \mathfrak{s}_2\}$$
$$\xrightarrow{\texttt{close}\ \mathfrak{s}_2}_t$$
$$\langle\!| (\texttt{join}\ y, [y \mapsto t_2, x \mapsto \ell])^t \cdot (\texttt{atomic}\ \{x[0] := x[0] + 1\}, [x \mapsto \ell])^{t_2}; \circ |\!\rangle^{\mathfrak{s}_1,t}_{\square}, [\ell \mapsto 2], \{\mathfrak{s}_1, \mathfrak{s}_2\}$$
$$\xrightarrow{\texttt{open}\ \mathfrak{s}_3}_{t2}$$
$$\langle\!| \begin{array}{l} (\texttt{join}\ y, [y \mapsto t_2, x \mapsto \ell])^t \cdot \\ (\texttt{atomic}\ \{x[0] := x[0] + 1\}, [x \mapsto \ell])^{t_2} \end{array} ; \langle\!| (x[0] := x[0] + 1, [x \mapsto \ell])^t; \circ |\!\rangle^{\mathfrak{s}_3,}_{\texttt{skip}} |\!\rangle^{\mathfrak{s}_1,t}_{\square}, \sigma \cdot [\ell \mapsto 2], \{\mathfrak{s}_1, \mathfrak{s}_2, \mathfrak{s}_3\}$$
$$\xrightarrow{\texttt{read}\ \ell\ 0\ 2}_t$$
$$\langle\!| (\texttt{join}\ y, [y \mapsto t_2, x \mapsto \ell])^t; \langle\!| (x[0] := 2 + 1, [x \mapsto \ell])^{t_2}; \circ |\!\rangle^{\mathfrak{s}_2,t}_{\texttt{skip}} |\!\rangle^{\mathfrak{s}_1,t}_{\square}, [\ell \mapsto 2], \{\mathfrak{s}_1, \mathfrak{s}_2, \mathfrak{s}_3\}$$
$$\xrightarrow{\texttt{write}\ \ell\ 0\ 3}_t$$
$$\langle\!| (\texttt{join}\ y, [y \mapsto t_2, x \mapsto \ell])^t; \langle\!| (\texttt{skip}, [x \mapsto \ell])^{t_2}; \circ |\!\rangle^{\mathfrak{s}_2,t}_{\texttt{skip}} |\!\rangle^{\mathfrak{s}_1,t}_{\square}, [\ell \mapsto 3], \{\mathfrak{s}_1, \mathfrak{s}_2, \mathfrak{s}_3\}$$
$$\xrightarrow{\texttt{close}\ \mathfrak{s}_3}_t$$
$$\langle\!| (\texttt{join}\ y, [y \mapsto t_2, x \mapsto \ell])^t \cdot (\texttt{skip}, [x \mapsto \ell])^{t_2}; \circ |\!\rangle^{\mathfrak{s}_1,t}_{\square}, [\ell \mapsto 3], \{\mathfrak{s}_1, \mathfrak{s}_2, \mathfrak{s}_3\}$$
$$\xrightarrow{\texttt{join}\ t_2}_t$$
$$\langle\!| (\texttt{skip}, [y \mapsto t_2, x \mapsto \ell])^t \cdot (\texttt{skip}, [x \mapsto \ell])^{t_2}; \circ |\!\rangle^{\mathfrak{s}_1,t}_{\square}, [\ell \mapsto 3], \{\mathfrak{s}_1, \mathfrak{s}_2, \mathfrak{s}_3\}$$

FIGURE 5.6 – *Exemple de réduction AFJ*

— $a = \texttt{fork}\ t'$: à partir de la règle de réduction (**fork**) appliquée à notre dernière réduction, nous avons l'hypothèse $fresh_{K',\phi'_0,\mathrm{B}''_0,t'_0}(t')$ (où $B'_0 = \langle\!|\ \phi'_0; \mathrm{B}''_0\ |\!\rangle^{\ t'_0}$). Nous savons que $fresh_{K,\phi_1,\mathrm{B}^\circ_1,t_0}(t')$ (où $B_1 = \langle\!|\ \phi_1; \mathrm{B}^\circ_1\ |\!\rangle^{\ t_0}$) n'est pas vérifié, puisque t' est ajouté à l'ensemble des processus légers. Les processus légers ne sont jamais enlevés de leur ensemble, chacun d'entre eux qui n'est pas frais à un moment, le restera pour tout futur état. Comme B'_0 peut être atteint à partir de B_1, et t' n'est pas frais dans B_1, nous pouvons conclure que t' n'est pas frais dans B'_0. Cela est en contradiction avec la règle (**fork**) appliquée à la dernière réduction.

— $a = \texttt{open}\ \mathfrak{s}$: à partir de la règle (**open**) nous avons $fresh_{S'_0}(\mathfrak{s})$ et $\neg fresh_{S_1}(\mathfrak{s})$. L'ensemble des noms de section ne diminue jamais durant une réduction. Cela signifie que si un nom de section n'est pas frais dans un certain état, il restera non frais pour tout futur état. Donc $\mathfrak{s}$ doit être non frais dans S'_0 puisque nous l'atteignons depuis S_1. Cela est en contradiction avec la règle (**open**) appliquée à la dernière réduction.

— $a = \texttt{close}\ \mathfrak{s}$: comme la section vient juste d'être fermée, nous avons $\mathfrak{s} \notin B_1$. Pour toutes bulle B atteignable depuis B_1 nous avons $\mathfrak{s} \notin B$ car les noms de sections sont uniques. Donc nous avons $\mathfrak{s} \notin B'_0$ puisque B'_0 peut être atteinte depuis B_1. Cependant la règle (**close**) appliquée à la dernière réduction nous permet de voir que $\mathfrak{s} \in B'_0$, cela nous amène donc à une contradiction.

□

Proposition 5.2 *Toute trace s produite par un programme p démarrant de l'état initial satisfait la condition* (**wf$_2$**) *:*

$$\forall i, \mathfrak{s}.\ \pi_s^{act}(i) = \texttt{close}\ \mathfrak{s} \Rightarrow$$
$$\exists j.\ j < i \wedge \pi_s^{act}(j) = \texttt{open}\ \mathfrak{s} \wedge \pi_s^{tid}(i) = \pi_s^{tid}(j).$$

Démonstration. Soit le nom de section $\mathfrak{s}$ et la position i telle que $\pi_s^{act}(i) = \texttt{close}\ \mathfrak{s}$. Nous devons trouver une position j telle que $j < i$, $\pi_s^{act}(j) = \texttt{open}\ \mathfrak{s}$ et $\pi_s^{tid}(j) = \pi_s^{tid}(i)$.

Nous procédons par induction sur la trace s. Le cas de base est trivial. Si nous avons $i < |s|$, nous pouvons utilisé notre hypothèse d'induction pour conclure. Maintenant nous supposons que $i = |s|$. Donc nous avons la dernière règle de réduction $K \vdash B_0, \sigma_0, P_0 \xrightarrow{\texttt{close}\ \mathfrak{s}}_t B_1, \sigma_1, S_1$, et nous pouvons affirmer que le nom de section $\mathfrak{s} \in B_0$.

Pour toute bulle B et programme p' et trace s' tels que ${}^{p'}B_{s'}$, et pour tout nom de section $\mathfrak{s}'$ tel que $\mathfrak{s}' \in B$, nous prouvons grâce à une induction sur la trace, qu'il existe une position k qui introduit ce nom de section par une réduction `open`.

Donc il existe une position j qui introduit ce nom de section par une réduction $K' \vdash B'_0, \sigma'_0, S'_0 \xrightarrow{\texttt{open}\ \mathfrak{s}}_{t'} B'_1, \sigma'_1, S'_1$.

Nous avons encore à prouver que $t' = t$. Pour chaque bulle, nous pouvons associer un nom de section à son processus léger propriétaire, et cette association ne change jamais. Donc nous avons $\mathfrak{s}$ associé à t' dans B'_1 et à t dans B_0, et donc nous avons $t' = t$. □

Proposition 5.3 *Toute trace s produite par un programme p partant de l'état initial satisfait la condition* (**wf$_3$**) *:*

$$\begin{gathered}\forall \mathfrak{s},\ i,\ j.\ range_s\ \mathfrak{s}\ (i,j) \Rightarrow \\ \pi_s^{act}(j) = \texttt{close}\ \mathfrak{s} \Rightarrow \\ \forall k,\ \mathfrak{s}'.\ i < k < j \Rightarrow \\ \pi_s^{tid}(i) = \pi_s^{tid}(k) \Rightarrow \\ \pi_s^{act}(k) = \texttt{open}\ \mathfrak{s}' \Rightarrow \\ \exists j',\ k < j' < j \wedge \pi_s^{act}(j') = \texttt{close}\ \mathfrak{s}'.\end{gathered}$$

Démonstration. Soient les positions i, i', j et les noms de section $\mathfrak{s}, \mathfrak{s}'$ tels que $\pi_s^{act}(i) = \texttt{open}\ \mathfrak{s}$, $\pi_s^{act}(i') = \texttt{open}\ \mathfrak{s}'$, $\pi_s^{tid}(i) = \pi_s^{tid}(i')$, $i < i' < j$ et $\pi_s^{act}(j) = \texttt{close}\ \mathfrak{s}$. Nous devons prouver qu'il existe une position j' telle que $\pi_s^{act}(j') = \texttt{close}\ \mathfrak{s}'$ et $i' < j' < j$.

Nous supposons qu'il existe une position j' telle que $range_s\ \mathfrak{s}'\ (i', j')$ et $j < j'$, et prouvons qu'il y a une contradiction.

Nous savons grâce au Lemme 5.1 que $cs(B^{i'})$ est un préfixe de $cs(B^{j-1})$ et $cs(B^i)$ est un préfixe de $cs(B^{i'})$.

Ensuite le lecteur peut prouver facilement que $cs(B^i) = cs(B^{j-1})$ avec (**open**) and (**close**). Donc nous avons que $cs(B^{i'})$ est un préfixe de $cs(B^i)$. Nous savons que $\mathfrak{s} \neq \mathfrak{s}'$ grâce à (**wf$_1$**), et donc $cs(B^{i'}) \neq cs(B^i)$. Ainsi $cs(B^{i'})$ ne peut pas être un préfixe de $cs(B^i)$ et $cs(B^i)$ être un préfixe de $cs(B^{i'})$.

□

Proposition 5.4 *Toute trace s produite par un programme p partant de l'état initial satisfait la condition* (**wf$_4$**) *:*

$$\begin{gathered}\forall i, t.\ \pi_s^{act}(i) = \texttt{fork}\ t \Rightarrow \\ \forall j.\ \pi_s^{tid}(j) = t \Rightarrow \\ i < j.\end{gathered}$$

Démonstration.

Soit l'identifiant de processus léger t et les positions i et j tels que $\pi_s^{act}(i) = \texttt{fork}\ t$ et $\pi_s^{tid}(j) = t$. Nous devons prouver que $i < j$. La preuve est faite par induction sur la trace s. La cas où la trace est vide est trivial. Soit les règles de réduction $K, \mathrm{B}_0^\circ, t_0 \vdash \phi_0, \sigma_0 \xrightarrow{\texttt{fork}\ t}_{t'} \phi_1, \sigma_1$, et $K', \mathrm{B}_0^{\circ\prime}, t_0' \vdash \phi_0', \sigma_0' \xrightarrow{c}_t \phi_1', \sigma_1'$ produisant respectivement les évènements à la position i et j.

Nous procédons en examinant la position de i, j dans la trace. Si $i < |s|$ nous utilisons notre hypothèse d'induction. Maintenant concentrons nous sur le cas $i = |s|$, et montrons que cela ne peut pas être possible.

— $j < |s|$: grâce à la règle (**fork**), nous savons que $fresh_{K,\phi_0,\mathrm{B}_0^\circ,t_0}(t)$ est vérifié et $fresh_{K',\phi_1',\mathrm{B}_0^{\circ\prime},t_0'}(t)$ ne l'est pas. Un processus léger qui n'est pas frais dans un certain état, le restera pour les états suivants. Puisque l'état de B_0° est atteignable depuis l'état de $\mathrm{B}_0^{\circ\prime}$, il y a une contradiction.

— $j = |s|$: le dernier évènement est $(t, \texttt{fork}\ t)$, un processus léger ne peut créer

un autre processus léger avec le même identifiant, cela n'est pas permis par le prédicat *fresh*

□

Proposition 5.5 *Toute trace s produite par un programme p partant de l'état initial satisfait la condition* (**wf**$_5$) *:*

$$\forall i, t.\ (\pi_s^{tid}(i) = t \ \vee\ \pi_s^{act}(i) = \texttt{fork}\ t) \Rightarrow \\ \forall j.\ \pi_s^{act}(j) = \texttt{join}\ t \Rightarrow \\ i < j.$$

Démonstration. Soit le processus léger t et les position i et j tels que $\pi_s^{tid}(i) = t$ ou $\pi_s^{act}(i) = \texttt{fork}\ t$, et $\pi_s^{act}(j) = \texttt{join}\ t$. Nous devons prouver que $i <$j. Soit les règles de réduction $K \vdash B_0, \sigma_0, S_0 \xrightarrow{a}_{t'} B_1, \sigma_1, S_1$ où soit $t' = t$ ou $a = \texttt{fork}\ t$, et $K' \vdash B'_0, \sigma'_0, S'_0 \xrightarrow{\texttt{join}\ t}_{t''} B'_1, \sigma'_1, S'_1$ produisant respectivement les évènements à la position i et j. La preuve est faite en examinant la position relative entre i et j. Si $i < j$ la résultat est immédiat. Pour les autres cas, nous devons prouver que nous arrivons à une contradiction.

— $i = j$: nous devons distinguer deux cas selon l'hypothèse que l'on a sur i :
  — $\pi_s^{tid}(i) = t$: nous pouvons affirmer que $B_0 = B'_0$ et $B_1 = B'_1$. La règle de réduction (**join**) énonce que le processus léger t doit vérifier $term_{K,\phi_0,\mathrm{B}_0^\circ,t_0}$ (où $B_0 = (\!|\ \phi_0; \mathrm{B}_0^\circ\ |\!)^{t_0}$), et ici ce n'est clairement pas le cas, puisque c'est la processus léger t qui fait l'action.
  — $\pi_s^{act}(i) = \texttt{fork}\ t$: il y a contradiction sur l'action effectuée.
— $i > j$: à partir de la règle (**join**) appliquée à la réduction associée à la position j, nous pouvons affirmer que $term_{K',\phi'_0,\mathrm{B}_0^{\circ\prime},t'_0}(t)$ (où $B'_0 = (\!|\ \phi'_0; \mathrm{B}_0^{\circ\prime}\ |\!)^{t'_0}$), et donc aucune action effectuée par de processus léger t ne peut être faite plus tard. De plus, t ne sera pas frais pour les prochaines bulles. Cela est en contradiction avec l'hypothèse sur la position i.

□

Proposition 5.6 *Toute trace s produite par un programme p partant de l'état initial satisfait la condition* (**wf**$_6$) *:*

$$\forall \mathfrak{s}, i, j, t.\ range_s\ \mathfrak{s}\ (i, j) \Rightarrow \\ owner_s\ \mathfrak{s}\ t \Rightarrow \\ \forall k.\ \pi_s^{act}(k) = \texttt{join}\ t \Rightarrow \\ j < k.$$

Démonstration.

Soit le nom de section $\mathfrak{s}$, les positions i, j, k et l'identifiant de processus léger t tel que $range_s\ \mathfrak{s}\ (i, j)$, $owner_s\ \mathfrak{s}\ t$ et $\pi_s^{act}(k) = \texttt{join}\ t$. Nous devons prouver que $j < k$.

Soit la règle de réduction $K_1, \mathrm{B}_1^\circ, t_1 \vdash \phi_1, \sigma_1 \xrightarrow{\texttt{join}\ t}_{t'} \phi_2, \sigma_2$ produisant l'évènement à la position k. Nous procédons par cas sur la forme de $range_s\ \mathfrak{s}\ (i, j)$.

— intervalle ouvert : Donc $j = |s|$ et alors le fait que $j < k$ est impossible. Nous devons prouver que nous avons une contradiction. Pour le faire, nous examinons la position relative entre k et i :
  — $i < k$: Pour toute bulle B atteignable depuis i, nous savons que $\mathfrak{s} \in B$ puisqu'il n'y a pas de `close` $\mathfrak{s}$, et ce nom de section est associé à l'identifiant de processus léger t car nous avons $owner_s\ \mathfrak{s}\ t$. À partir de la règle (**join**), nous déduisons que t ne possède pas de section. Cela nous amène à une contradiction.
  — $i > k$: à partir de la règle (**join**), nous savons que l'instruction associée à t est `skip`. Cette association ne changera jamais. Cependant nous avons également $\pi_s(i) = (t, \texttt{open}\ \mathfrak{s})$, donc t est associé à l'instruction `atomic`. Cela nous amène à une contradiction.
— intervalle fermé: supposons que $k < j$ et nous prouvons que nous en déduisons deux conclusions en contradiction, à partir des deux règles suivantes :
  — à partir de la règle (**join**), nous savons que $term_{K_1,\phi_1,\mathrm{B}_1^\circ,t_1}(t)$, et donc que t n'est le propriétaire d'aucune section à ce moment. Comme t est associé à l'instruction `skip`, t ne sera pas le propriétaire d'aucune section dans le futur.
  — à partir de la règle (**close**), nous savons que t est le propriétaire de la section $\mathfrak{s}$.

  Ces deux conclusions sont en contradiction, donc $j < k$.

□

Proposition 5.7 *Toute trace s produite par un programme p partant de l'état initial satisfait la condition* (**wf**$_7$) *:*

$$\forall t, i, j.\ \pi_s^{act}(i) = \texttt{fork}\ t \Rightarrow \pi_s^{act}(j) = \texttt{join}\ t \Rightarrow see_s\ i\ j.$$

Démonstration. Soit les positions i, j et l'identifiant de processus léger t tel que $\pi_s^{act}(i) = \texttt{fork}\ t$ et $\pi_s^{act}(j) = \texttt{join}\ t$. Nous devons prouver $see_s\ i\ j$. Soit l'identifiant de processus léger t_0 tel que $\pi_s^{act}(j) = (t_0, \texttt{join}\ t)$.

Premièrement, nous notons $t' \in_B t$ si $t \in B$ et $\exists x,\ \rho_t(x) = t'$ où ρ_t est l'environnement local de t. Nous utilisons la définition suivante

$$\begin{aligned} seeFork_s\ t\ t'\ j \quad = \quad & (\exists i,\ \pi_s^{act}(i) = \texttt{fork}\ t' \ \wedge\ (\pi_s^{tid}(j) = t \ \vee\ \pi_s^{act}(j) = \texttt{fork}\ t) \ \wedge\ see_s\ i\ j) \\ & \vee\ (\pi_s(j) = (t, \texttt{fork}\ t')) \end{aligned}$$

Avant de prouver notre résultat, nous prouvons un résultat intermédiaire énonçant que pour toute bulle B atteignable par l'exécution d'un programme qui génère une trace s, et pour tous processus légers t, t' tels que $t' \in_B t$, qu'il existe une position j telle que $seeFork_s\ t\ t'\ j$. Nous le prouvons par induction généralisée sur la trace s.

Soit $s \cdot a$ la nouvelle trace, et soit B la bulle atteinte par cette trace. Nous savons par hypothèse que $t' \in_B t$. Nous devons prouver qu'il existe une position j telle que $seeFork_{s\cdot a}\ t\ t'\ j$.

Soit la bulle B' atteignable par la trace s. Si $t' \in_{B'} t$ est vérifiée, nous pouvons appliquer l'hypothèse d'induction. Il existe donc une position j dans la trace s et donc dans $s \cdot a$, telle que $seeFork_{s \cdot a}\ t\ t'\ j$.

Si $t' \notin_{B'} t$, nous devons déterminer quelle est l'instruction qui fait que $t' \in_B t$ est valide. Nous devons raisonner par cas sur la dernière action effectuée. Nous considérons donc uniquement les règles susceptibles de modifier l'environnement local en écrivant une valeur de type identifiant de processus léger. Quatre règles peuvent modifier l'environnement local. La règle (**assign**) est capable de le modifier, mais elle ne doit pas être prise en compte, car une expression ne peut être évaluée en un nouveau processus léger. En effet à cause de la restriction sur les constantes utilisateurs, l'expression ne peut pas être une constante identifiant de processus léger. L'expression ne peut pas être non plus une variable contenant l'identifiant du processus, cela serait en contradiction avec l'hypothèse $t' \notin_{B'} t$. La règle (**alloc**) n'est pas non plus envisageable car elle retourne une nouvelle adresse. Donc la dernière action est faite soit par la règle (**get**) ou par la règle (**fork**). Il a donc soit un lecture dans le tas qui récupère l'identifiant t', plus formellement, il existe ℓ, n tel que $(t, \texttt{read}\ \ell\ n\ t')$, ou alors cela provient de la création d'un nouveau processus léger. Il existe deux possibilités pour ce dernier cas. Il existe un processus léger t_0 tel que $t' \in_{B'} t_0$, et créant le processus léger t en lui donnant comme paramètre t'. La deuxième possibilité est que le processus léger t crée t'. Examinons ces trois cas :

— $(t, \texttt{read}\ \ell\ n\ t')$: si une valeur est lue, elle a été nécessairement écrite avant, donc il existe une position k_1 et un processus léger t_1 tel que $\pi_{s\cdot(t,\texttt{read}\ \ell\ n\ t')}(k_1) = (t_1, \texttt{write}\ \ell\ n\ t')$. Soit B_1 la bulle atteinte après la position k_1. Nous savons que $t' \in_{B_1} t_1$. Nous pouvons donc appliquer l'hypothèse d'induction. Il existe donc une position j_0 telle que $seeFork_{s\cdot(t,\texttt{read}\ \ell\ n\ t')}\ t_1\ t'\ j_0$. Nous raisonnons par cas suivant les deux cas de la disjonction de cette hypothèse:
  — Soit la position i_0 telle que $\pi^{act}_{s\cdot(t,\texttt{read}\ \ell\ n\ t')}(i_0) = \texttt{fork}\ t'$. Par la première partie de la disjonction de l'hypothèse d'induction nous savons aussi que $(\pi^{tid}_{s\cdot(t,\texttt{read}\ \ell\ n\ t')}(j_0) = t_1 \vee \pi^{act}_{s\cdot(t,\texttt{read}\ \ell\ n\ t')}(j_0) = \texttt{fork}\ t_1) \wedge see_{s\cdot(t,\texttt{read}\ \ell\ n\ t')}\ i_0\ j_0$.
  Ensuite nous voulons prouver que $see_{s\cdot(t,\texttt{read}\ \ell\ n\ t')}\ i_0\ |s|$. Nous avons par définition $see_{s\cdot(t,\texttt{read}\ \ell\ n\ t')}\ j_0\ k_1$ et $see_{s\cdot(t,\texttt{read}\ \ell\ n\ t')}\ k_1\ |s|$. Nous avons donc $see_{s\cdot(t,\texttt{read}\ \ell\ n\ t')}\ i_0\ j_0$, $see_{s\cdot(t,\texttt{read}\ \ell\ n\ t')}\ j_0\ k_1$ et $see_{s\cdot(t,\texttt{read}\ \ell\ n\ t')}\ k_1\ |s|$. Nous avons donc bien par transitivité $see_{s\cdot(t,\texttt{read}\ \ell\ n\ t')}\ i_0\ |s|$. Comme nous avons également $\pi^{tid}_{s\cdot(t,\texttt{read}\ \ell\ n\ t')}(|s|) = t$, nous avons bien $seeFork_{s\cdot(t,\texttt{read}\ \ell\ n\ t')}\ t\ t'\ |s|$
  — La deuxième partie de la disjonction de l'hypothèse d'induction nous indique $\pi_{s\cdot(t,\texttt{read}\ \ell\ n\ t')}(j_0) = (t_1, \texttt{fork}\ t')$, et l'on sait que $j_0 < k_1$ car nous avons eu l'hypothèse d'induction sur la trace allant jusqu'à la position k_1. Nous avons $see_{s\cdot(t,\texttt{read}\ \ell\ n\ t')}\ j_0\ k_1$ car les deux actions sont faites par le même processus léger. Nous avons également $see_{s\cdot(t,\texttt{read}\ \ell\ n\ t')}\ k_1\ |s|$ par définition. Nous avons donc par transitivité $see_{s\cdot(t,\texttt{read}\ \ell\ n\ t')}\ j_0\ |s|$. Comme nous avons également $\pi^{tid}_{s\cdot(t,\texttt{read}\ \ell\ n\ t')}(|s|) = t$, nous avons bien $seeFork_{s\cdot(t,\texttt{read}\ \ell\ n\ t')}\ t\ t'\ |s|$

— $(t_0, \texttt{fork}\ t) \wedge t' \in_{B'} t_0$: Grâce à $t' \in_{B'} t_0$ nous pouvons appliquer l'hypothèse d'induction. Nous savons donc qu'il existe une position j_0 telle que $\mathit{seeFork}_{s\cdot(t_0,\texttt{fork}\ t)}\ t_0\ t'\ j_0$. Nous considérons les deux cas de la disjonction :
  — il existe une position i_0 telle que $\pi^{act}_{s\cdot(t_0,\texttt{fork}\ t)}(i_0) = \texttt{fork}\ t' \wedge \mathit{see}_{s\cdot(t_0,\texttt{fork}\ t)}\ i_0\ j_0$. Par définition nous avons $\mathit{see}_{s\cdot(t_0,\texttt{fork}\ t)}\ j_0\ |s|$, et donc par transitivité $\mathit{see}_{s\cdot(t_0,\texttt{fork}\ t)}\ i_0\ |s|$. Comme nous avons également $\pi^{act}_{s\cdot(t_0,\texttt{fork}\ t)}(|s|) = \texttt{fork}\ t$ nous avons bien $\mathit{seeFork}_{s\cdot(t_0,\texttt{fork}\ t)}\ t\ t'\ |s|$.
  — $\pi_{s\cdot(t_0,\texttt{fork}\ t)}(j_0) = (t_0, \texttt{fork}\ t')$. On sait que l'on a $\mathit{see}_{s\cdot(t_0,\texttt{fork}\ t)}\ j_0\ |s|$ par définition. Nous avons donc $\pi^{act}_{s\cdot(t_0,\texttt{fork}\ t)}(j_0) = \texttt{fork}\ t' \wedge \pi^{act}_{s\cdot(t_0,\texttt{fork}\ t)}(|s|) = \texttt{fork}\ t$. Nous avons donc bien $\mathit{seeFork}_{s\cdot(t_0,\texttt{fork}\ t)}\ t\ t'\ |s|$

— $(t, \texttt{fork}\ t')$: Le résultat est immédiat, nous avons bien $\mathit{seeFork}_{s\cdot(t,\texttt{fork}\ t')}\ t\ t'\ |s|$.

Nous utilisons ce dernier résultat pour prouver notre proposition. Nous savons que nous avons $t \in_B t_0$ grâce à (**join**), donc il existe une position k telle que $\mathit{seeFork}_s\ t_0\ t\ k$. Examinons les deux cas de la disjonction de cette hypothèse :

— Il existe une position i_0 telle que $\pi^{act}_s(i_0) = \texttt{fork}\ t \wedge (\pi^{tid}_s(k) = t_0 \vee \pi^{act}_s(k) = \texttt{fork}\ t_0) \wedge \mathit{see}_s\ i_0\ k$. Or d'après la proposition 5.1, nous pouvons conclure que $i_0 = i$. Pour les deux cas de la disjonction $(\pi^{tid}_s(k) = t_0 \vee \pi^{act}_s(k) = \texttt{fork}\ t_0)$, nous pouvons conclure que $\mathit{see}_s\ k\ j$. Et donc nous avons bien $\mathit{see}_s\ i\ j$.
— Nous savons que $\pi^{act}_s(k) = (t_0, \texttt{fork}\ t)$. Par le proposition 5.1, nous savons que $k = i$. Nous avons bien $\mathit{see}_s\ i\ j$ par définition.

□

Proposition 5.8 *Toute trace s produite par un programme p partant de l'état initial, satisfait la condition* (**wf**$_8$) *:*

$$\forall \mathfrak{s}, \mathfrak{s}',\ \texttt{open}\ \mathfrak{s} \in s \Rightarrow \texttt{open}\ \mathfrak{s}' \in s \Rightarrow \mathfrak{s} \smile_s \mathfrak{s}' \Rightarrow \mathfrak{s} \prec_s \mathfrak{s}' \vee \mathfrak{s}' \prec_s \mathfrak{s}.$$

Démonstration. Soit les positions i, i', j, j', les processus légers t, t' et les noms de sections $\mathfrak{s}, \mathfrak{s}'$ tels que $\mathit{range}_s\ \mathfrak{s}\ (i, j)$, $\mathit{range}_s\ \mathfrak{s}'\ (i', j')$, et t, t' sont les propriétaires respectifs de $\mathfrak{s}, \mathfrak{s}'$. Nous devons prouver que $\mathfrak{s} \prec_s \mathfrak{s}'$ ou $\mathfrak{s}' \prec_s \mathfrak{s}$, autrement dit $j < i' \vee j' < i$. Nous raisonnons pas cas sur la position relative de i', i, j, j'. Premièrement nous savons que $i \neq i'$, sinon $\mathfrak{s} = \mathfrak{s}'$, ce qui est en contradiction avec $\mathfrak{s} \smile_s \mathfrak{s}'$. Nous savons également que $i \leq j$ et $i' \leq j'$ grâce à (**wf**$_2$) et (**wf**$_1$). Si $j < i'$ ou $j' < i$ nous pouvons conclure. Les cas problématiques ont lieu lorsque $i < i' \leq j$ ou $i' < i \leq j'$. Ces deux cas étant symétriques, nous examinons seulement le premier. Donc nous supposons que $i < i' \leq j$ et nous prouvons que cela nous amène à une contradiction. Nous avons que $cs(B^i)$ est un préfixe de $cs(B^{i'})$ par le Lemme 5.1. Grâce à (**open**) nous prouvons que le dernier élément de $cs(B^i)$ est (t, p) et le dernier de $cs(B^{i'})$ est $(t', \mathfrak{s}')$. Ainsi nous avons $\mathit{tribe}_s\ \mathfrak{s}\ t'$ par le Lemme 5.3. Alors nous prouvons que nous avons $\mathfrak{s}' \Subset_s \mathfrak{s}$ par définition de $\Subset$. Cette dernière conclusion est en contradiction avec notre hypothèse $\mathfrak{s} \smile_s \mathfrak{s}'$. □

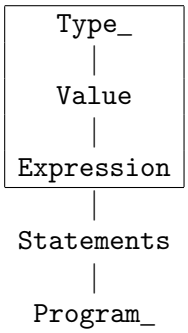


FIGURE 5.7 – *Organisation générale de la formalisation en Coq de AFJ*

Théorème 5.1 *Toute trace produite par un programme vérifie les conditions de bonne-formation.*

Démonstration. Par les Propositions (5.1) à (5.8). □

À partir du chapitre précédent (chapitre 4), nous savons que les traces bien formées et bien synchronisées sont équivalentes à une trace sérielle. Donc comme la sémantique opérationnelle AFJ produit des traces bien formées, nous savons que toutes les exécutions de programmes AFJ bien synchronisées produisent des traces équivalentes à une trace sérielle.

5.4 Formalisation en Coq

5.4.1 Organisation générale

L'organisation des fichiers suit les règles définies dans le chapitre précédent. L'apport de ce chapitre est la définition de la syntaxe et de la sémantique de programmes AFJ, ainsi que la preuve que cette sémantique produit des traces bien formées. Ils se retrouvent principalement dans les fichiers liés à `Program` et `Statements`. Les définitions des valeurs et des expressions, partagées avec le reste du développement, sont utilisées. Dans la figure 5.7, les bibliothèques communes à plusieurs langages sont encadrées.

La syntaxe du programme ainsi que les règles de réduction de processus légers sont définis dans `Statements.v`, les règles de réductions inter processus légers et les règles de réduction de sections atomiques sont définies dans `Program.v`. Chacune des preuves des conditions de bonne formation sont définies dans des fichiers dont le nom est composé de `Program_Theory_execution_wf` et du nom de la condition. Toutes ces preuves sont réunis dans le fichier `Program_Theory.v`.

De manière similaire aux chapitres précédents, les définitions et les lemmes sont encapsulés dans des modules. Lorsque nous définissons le type principal dans ces modules, nous prenons comme convention de le nommer *t*.

5.4.2 Choix de formalisation

Un type de module Type_.TYPE regroupe la définition des types manipulables par les programmes AFJ ainsi qu'une fonction de traduction et quelques propriétés. Les types (Thread, Number, Boolean, Location)) des valeurs qui sont manipulées dans les programmes sont définis dans un type inductif. Une fonction typeOfType traduit ces constructeurs en types Coq. Certaines traductions sont immédiates : Boolean vers bool, Number vers Z. Nous avons choisi de traduire les Thread par des nat. Les location sont du type abstrait Address.t infini dénombrable avec égalité décidable.

Le type de module Value.TYPE concerne les valeurs elles-mêmes, ces dernières étant typées. L'inductif définissant les valeurs AFJ ne contient qu'un seul constructeur car défini à l'aide d'un type dépendant (ici Ty est un module dont le type est Type_.TYPE) :

```
Inductive t := | Value : ∀(ty:Ty.t)(v:Ty.typeOfType ty), t.
```

Le type de module Expression traite des expressions. Comme définies plus haut, les expressions peuvent être des constantes, des variables ou des applications d'opération. Le type suivant pourrait refléter cette définition :

```
Inductive t : Type :=
| Const : Va.t →t
| Var : nat →t
| Ope : operation →list t →t.
```

où Va est un module de type Value.TYPE. Toutefois cette définition des termes n'est pas conforme à la définition donnée dans la section 5.1. En effet nous avons indiqué que les identifiants de processus léger ne peuvent pas être écrits en tant que constantes par l'utilisateur mais seulement générées par l'instruction de création de processus léger. Nous introduirons une restriction supplémentaire pour le langage du chapitre 6. Afin de modéliser correctement ceci, nous avons fait le choix d'avoir le type suivant pour les expressions :

```
Inductive t : Type :=
| Var : nat →t
| Ope : operation →list t →t.
```

et de contrôler plus finement les constantes autorisée via des opérations prédéfinies « constantes ».

Les opérations sont définies par un type inductif ayant essentiellement des constructeurs constants, sauf pour ce qui est de l'égalité qui est typée, le type des éléments comparés est ainsi un argument du constructeur, et les opérations qui sont là pour écrire des constantes qui prennent en argument la valeur de la constante. Une fonction associe une signature à chacune de ces opérations :

```
Module Signature. Module Type T.
  Inductive domain : Set :=
    | Total: ∀(t:Ty.t), domain
```

```
    | Partial : ∀(t:Ty.t), domain.
  Definition t := list Ty.t * domain.
End T. End Signature.
```

Pour une opération constante, la première composante est la liste vide. Le type de retour d'une opération prédéfinie de nos sémantiques peut être total ou partiel. De la même façon qu'un type AFJ est traduit en type Coq, une signature d'une opération prédéfinie de AFJ est traduite en un type Coq de fonction par la fonction typeOf du type de module Signature.T. Par exemple la traduction de la signature ([Number; Number], Total Number) de l'opération Plus est Z →Z →Z, mais la traduction de la signature ([Number; Number], Partial Number) de l'opération Div est Z →Z →result Z. La signature de l'opération constante entière est ([],Total Number).

Le type result permet de renvoyer une valeur ou bien un message d'erreur. Par exmple il permet de retourner un message d'erreur si nous tentons de faire une division par zéro. L'évaluation d'une expression prend en paramètre un environnement local et une expression et renvoie également une valeur de type result.

Les environnements locaux sont définis abstraitement dans un type de module Store.T :

```
Module Type T (Id Ad : DecidableInfiniteSet) (Ty : Type_.TYPE Ad) (Va : Value.TYPE Ad Ty).
  Parameter t : Type.
  Parameter empty : t.
  Parameter set : t →Id.t →Vl.t →t.
  Parameter get : t →Id.t →option Vl.t.
  Axiom get_empty : ∀id, get empty id = None.
  Axiom get_set_same : ∀store id val, get (set store id val) id = Some val.
  Axiom get_set_diff : ∀store id id' val, ∼id=id' →get (set store id val) id' = get store id'.
End T.
```

Le tas est quant à lui représenté par une fonction :

```
Definition heap : Set := Ad.t →option ( list Va.t ) .
```

Une des notions les plus importantes définis dans ce chapitre, est la notion de bulle. Comme c'est une définition récursive, elle est nécessairement représentée par un type inductif. Comme la sous-bulle peut être présente ou non, nous utilisons le type option pour la représenter.

```
  Inductive Bubble :=
  | B_atomic : Threads →option Bubble →sectionName →threadId →context →Bubble.
```

5.4.3 Correspondance

Le développement en Coq est disponible l'adresse suivante :

`https://traclifo.univ-orleans.fr/PaPDAS/wiki/TransactionsInCoq`

Les tableaux qui suivent donnent la correspondance entre les définitions (figure 5.8) et les lemmes (figure 5.9) et leurs équivalents en Coq. Pour chacun d'entre eux nous indiquons son nom en Coq et le fichier. Tous les fichiers sont dans les répertoires `Common` ou `Transactions` de l'archive.

Nom	Coq	Fichier
Règles de réduction intra-thread	reduce_intra	`Statements.v`
Règles de réduction inter-thread	reduce_inter	`Program_.v`
Règles de réduction de sections atomiques	reduce_atomic	`Program_.v`
Règle de réduction principale	reduce	`Program_.v`
Valeur	t (dans le module TYPE))	`Value.v`
Expression	t (dans le module TYPE))	`Expression.v`
Commandes	t (dans le module TYPE))	`Statements.v`
Bulle	Bubble (dans le module TYPE))	`Statements.v`
Etat	state (dans le module TYPE))	`Statements.v`

FIGURE 5.8 – *Table de correspondance avec les définitions Coq*

Nom	Coq	Fichier	Terminé
Proposition 5.1	execution_wf_occurences	`Program_Theory_execution_wf_occurences.v`	Oui*
Proposition 5.2	execution_wf_open_close	`Program_Theory_execution_wf_open_close.v`	Oui
Proposition 5.3	execution_wf_seq_order	`Program_Theory_execution_wf_seq_order.v`	Non
Proposition 5.4	execution_wf_fork	`Program_Theory_execution_wf_fork.v`	Oui
Proposition 5.5	execution_wf_join	`Program_Theory_execution_wf_join.v`	Non*
Proposition 5.6	execution_wf_join_all_closed	`Program_Theory_execution_wf_join_all_closed.v`	Non
Proposition 5.7	execution_wf_see_fork	`Program_Theory_execution_wf_join_see_fork.v`	Non*
Proposition 5.8	execution_wf_mutual_exclusion	`Program_Theory_wf_mutual_exclusion.v`	Non
Théorème 5.1	execution_wellFormed	`Program_Theory.v`	Oui

(Oui* : preuve où il reste quelques manques, Non*: structure de la preuve présente)

FIGURE 5.9 – *Table de correspondance avec les énoncés et preuves Coq*

5.A Preuves des lemmes

Lemme 5.1 *Pour tout programme p, trace s, positions i, j, nom de section $\mathfrak{s}$, tels que s est la trace générée par l'exécution du programme p partant de l'état initial et $range_s\ \mathfrak{s}\ (i,j)$, nous avons pour toute position k telle que $i \leq k < j$ si c'est un intervalle fermé, ou $i \leq k \leq j$ si c'est un intervalle ouvert, que la séquence caractéristique $cs({}^pB_s^i)$ est un préfixe de $cs({}^pB_s^k)$.*

Démonstration. La preuve est faire par induction sur k. Pour le cas de base où $k = 0$, si $i = 0$ alors $cs({}^pB_s^i) = cs({}^pB_s^k)$ et donc la relation préfixe est vérifiée, sinon si $i \neq 0$ alors l'hypothèse $i \leq k$ est contredite.

Pour le cas inductif, si $k = i$ alors $cs(B^i) = cs(B^k)$ et donc la relation préfixe est vérifiée. Sinon, puisque $i < k < j$, nous avons $i \leq (k-1) < j$, et donc l'hypothèse d'induction peut être utilisée.

Nous avons alors besoin de raisonner sur le type d'action faite en k. Si l'action n'est pas un `open` ou un `close`, la séquence caractéristique n'est pas modifiée, donc la relation préfixe est toujours vérifiée.

- Supposons que l'action est faite en k est `open`. Il existe une paire r telle que $cs(B^k) = cs(B^{k-1}) \cdot r$. Donc $cs(B^{k-1})$ est un préfixe de $cs(B^k)$ et alors par transitivité $cs(B^i)$ est un préfixe de $cs(B^k)$.
- Supposons que l'action est faite en k est `close`. Il existe donc un nom de section atomique $\mathfrak{s}'$ tel que $\pi_s^{act}(k) = \texttt{close}\ \mathfrak{s}'$. Nous examinons si $cs(B^i)$ est un préfixe strict de $cs(B^{k-1})$ ou non.
  - $cs(B^i) = cs(B^{k-1})$: dans ce cas, nous avons par (**close**) que $\mathfrak{s}' = \mathfrak{s}$. Maintenant voyons quel intervalle nous avons:
    - intervalle fermé: On sait donc que $\pi_s^{act}(j) = \texttt{close}\ \mathfrak{s}$. Comme l'on sait que $k \neq j$, il existe donc deux actions `close` $\mathfrak{s}$ dans la trace. Cela est en contradiction avec ($\mathbf{wf_1}$).
    - intervalle ouvert: L'existence de $\pi_s^{act}(k) = \texttt{close}\ \mathfrak{s}$ est en contradiction avec la définition d'intervalle ouvert.
  - $cs(B^i)$ est un préfixe stricte de $cs(B^{k-1})$, et grâce à (**close**), nous savons qu'il existe une paire r telle $cs(B^k) \cdot r = cs(B^{k-1})$. Donc $cs(B^i)$ est un préfixe de $cs(B^k)$.

□

Lemme 5.3 *Pour tout programme p, trace s, bulle B tels que $B = {}^pB_s$, et pour toute séquence l telle que l est un préfixe de $cs(B)$, alors pour tout $(t, \mathfrak{s}) \in l$ et pour tout $(t', \mathfrak{s}') \in cs(B)$ et $(t', \mathfrak{s}') \notin l$, nous avons $tribe_s\ \mathfrak{s}\ t'$.*

Démonstration. La preuve est faite par induction sur la trace s. La cas de base est trivial. Ensuite nous avons notre hypothèse d'induction sur B_s, nous devons le prouver pour $B_{s\cdot(t_1,a_1)}$. Nous examinons l'action a_1. Si ce n'est pas `close` ou un `open`, $cs(B_s) = cs(B_{s\cdot(t_1,a_1)})$, et nous pouvons conclure. Examinons les autres cas:

— $a_1 = \texttt{close}\ \mathfrak{s}''$: nous savons grâce à la règle (**close**), qu'il existe une paire b telle que $cs(B_{s\cdot(t_1,a_1)}) \cdot b = cs(B_s)$, et donc que chaque liste qui est préfixe de $cs(B_{s\cdot(t_1,a_1)})$ est aussi un préfixe de $cs(B_s)$, donc nous pouvons utiliser notre hypothèse d'induction pour conclure.
— $a_1 = \texttt{open}\ \mathfrak{s}''$: soit $(t_0, \mathfrak{s}_0)$ le dernier élément de $cs(B_s)$ (si une telle paire n'existe pas, alors $cs(B_{s\cdot(t_1,a_1)}) = (t_1, a_1)$ et il est facile de conclure). $tribe_{s\cdot(t_1,a_1)}\ \mathfrak{s}_0\ t_1$ est vérifiée grâce à la règle (**open**). Donc nous pouvons conclure grâce à l'hypothèse d'induction, et par la définition de *tribe*.

□

LUFJ: un langage à processus légers et verrous

6

Sommaire

Le chapitre précédant présentait AFJ, un langage impératif avec processus légers dont le mécanisme de synchronisation principal est la section atomique. Nous considérons dans ce chapitre le langage *Lock Unlock Fork Join* ou LUFJ. Le noyau impératif et de gestion de processus légers est le même que AFJ mais ici le mécanisme de synchronisation est basé sur les verrous.

6.1 Le langage *Lock Unlock Fork Join*

Le langage *Lock Unlock Fork Join* (LUFJ) est identique au langage *Atomic Fork Join* (AFJ) excepté pour les primitives de synchronisation : il n'y a plus de primitive de sections atomiques, mais il est possible d'utiliser des verrous comme mécanisme de synchronisation.

Afin de faciliter la lecture, nous rappelons les éléments communs aux deux langages. Nous supposons des ensembles dénombrables disjoints d'emplacements mémoire, de noms de processus léger, dont les éléments sont notés respectivement ℓ et t, éventuellement avec des indices. Il y a également un emplacement mémoire particulier noté `null` qui ne peut être alloué. L'ensemble des valeurs, dont les éléments sont notés

v, éventuellement avec des indices, contient au moins les emplacements mémoires, les entiers, les booléens et les noms de processus légers.

Il n'y a pas de valeurs spécifiques pour les verrous. Nous utilisons les emplacements mémoire en tant qu'identifiants de verrou. Dans la sémantique opérationnelle toutefois l'état des verrous n'est pas présent dans le tas. Nous ajoutons aux états une fonction partielle des adresses mémoires vers des valeurs encodant les états des verrous. Ce choix a essentiellement été guidé par une volonté de réutiliser la modélisation Coq des valeurs de AFJ pour qu'elle soit commune aux deux langages.

Dans les grammaires suivantes, $\mathcal{X}$ signifie l'ensemble dénombrable des variables locales, $\bar{e}$ est un tuple d'expressions et `op` indique une opération prédéfinie sur les valeurs. Nous distinguons les valeurs pouvant être écrire en tant que constantes par l'utilisateur d, et l'ensemble des valeurs pouvant être produites par un programme v.

$$\begin{array}{lll} d & ::= & n \mid b \mid \texttt{null} \quad \text{où } n \in \mathbb{Z} \text{ et } b \in \mathbb{B} \\ e & ::= & d \mid x \mid \texttt{op}(\bar{e}) \quad \text{où } x \in \mathcal{X} \end{array}$$

La grammaire pour les instructions de LUFJ est la suivante :

$$\begin{array}{lll} c & ::= & \mid x := e \mid x := y[e] \mid x[e] := e' \\ & & \mid x := \texttt{allocate}(e) \mid \texttt{dispose}(e) \\ & & \mid \texttt{if } e \texttt{ then } b \texttt{ else } b \\ & & \mid \texttt{while } e \texttt{ do } b \\ & & \mid x = \texttt{fork}(m, e) \mid \texttt{join } e \\ & & \mid x := \texttt{init}() \\ & & \mid \texttt{lock } x \mid \texttt{unlock } x \mid \texttt{mlock } \overline{instr_{rl}}\ e \end{array}$$

La grammaire $instr_{rl}$ regroupe les instructions de lecture et de verrouillage utilisables dans l'instruction `mlock`.

$$instr_{rl} \quad ::= \quad x := y[e] \mid \texttt{lock } x$$

Ces instructions sont organisées en blocs, qui sont des listes d'instructions :

$$b \quad ::= \quad [] \mid c; b$$

Le corps d'une méthode et le corps principal d'un programme sont maintenant des blocs :

$$\begin{array}{rcl} mth & ::= & \overline{m(x)\ b} \\ program & ::= & \overline{mth}\ b \end{array}$$

Le sous-langage impératif séquentiel de LUFJ est identique à celui de AFJ et est très classique. Au niveau des instructions pour le parallélisme, nous retrouvons la création et l'attente de processus légers (`fork` et `join`). D'autre part nous ajoutons des instructions pour la manipulation de verrous. Les verrous sont stockables et créés dynamiquement (`init`). Comme d'habitude un verrou peut être acquis (`lock`) et relâché (`unlock`). L'instruction `lock` bloque le reste de l'exécution du processus léger, si le verrou est déjà pris. Une instruction moins classique est l'instruction `mlock` qui prend en

argument une liste d'instructions appartenant à $instr_{rl}$ et une expression (booléenne). La liste d'instructions contient des instructions permettant de lire la mémoire, afin de récupérer des verrrous, et des instructions de prises des verrous. La condition bloque l'exécution du processus léger tant qu'elle n'est pas vérifiée. Cette instruction représente une forme très restreinte de l'atomicité car réduit à un petit nombre d'instructions, et facilite les preuves. Cette instruction n'est pas irréaliste, et nous discutons de son implantation dans la section 6.2.3.

6.2 Une sémantique opérationnelle

6.2.1 États, actions et évènements

Les états de LUFJ reprennent les éléments suivants des états de AFJ. Chaque processus léger a son propre environnement local. Les processus légers se partagent le tas. L'environnement local est une fonction partielle des variables vers les valeurs. Le tas est une fonction partielle prenant un emplacement mémoire et un décalage et retournant une valeur. La valeur d'une cellule mémoire non-allouée est indéfinie. L'ensemble des processus légers est une fonction partielle des identifiants de processus légers vers une paire constituée du code restant à être exécuté par le processus léger, et son environnement local. $\mathcal{V}$ et $\mathcal{C}$ dénotent respectivement l'ensemble des valeurs et instructions. Les états de LUFJ contiennent un élément supplémentaire : une table des verrous. Un identifiant de verrou est associé soit au nom de processus léger qui l'a pris, soit à la valeur $\bullet$ qui indique que le verrou n'est pas pris. Nous utilisons les mêmes notations pour le tas et la table de verrous.

$$\begin{array}{lcll} \rho & \in & \mathcal{E} = \mathcal{X} \rightharpoonup \mathcal{V} & \text{environnent local} \\ \sigma & \in & \mathbb{L} \rightharpoonup \mathbb{N} \rightharpoonup \mathcal{V} & \text{tas} \\ \phi & \in & \mathbb{T} \rightharpoonup \mathcal{C} \times \mathcal{E} & \text{ensemble de processus légers} \\ \Lambda & \in & \mathbb{L} \rightharpoonup \mathbb{T} \cup \{\bullet\} & \text{table des verrous} \end{array}$$

L'état d'un programme LUFJ en cours d'exécution est un triplet ϕ, σ, Λ.

Les actions pour LUFJ sont similaires à celles de AFJ, à l'exception des actions liées aux sections atomiques qui disparaissent et des actions liées aux verrous qui sont ajoutées. L'action `init` ℓ indique qu'un verrou a été crée à l'emplacement ℓ. Les actions `lock` ℓ, `unlock` ℓ, et `mlock` $\overline{\ell}$ dénotent respectivement le verrouillage du verrou ℓ, le déverrouillage du verrou ℓ, et le verrouillage multiple de l'ensemble de verrous $\overline{\ell}$.

$$\begin{array}{lcl} a & ::= & \mid \tau \\ & & \mid \texttt{alloc}\ \ell\ n \mid \texttt{free}\ \ell \mid \texttt{read}\ \ell\ n\ v \mid \texttt{write}\ \ell\ n\ v \\ & & \mid \texttt{fork}\ t \mid \texttt{join}\ t \mid \texttt{init}\ \ell \mid \texttt{lock}\ \ell \mid \texttt{unlock}\ \ell \mid \texttt{mlock}\ \overline{\ell} \end{array}$$

Un évènement est un couple (t, a) d'identifiant de processus léger et d'action.

$$\vdash_p (x := e; b, \rho)^t,\ \sigma \xrightarrow{\tau}_t (b, \rho[x \mapsto v])^t,\ \sigma \quad \textbf{(assign)}$$
$$\text{si } [\![e]\!]_\rho = v$$

$$\vdash_p (x := \texttt{allocate}(e); b, \rho)^t,\ \sigma \xrightarrow{\texttt{alloc}\ \ell\ n}_t (b, \rho[x \mapsto \ell])^t,\ \sigma \cdot [\ell \mapsto [0]^n] \quad \textbf{(alloc)}$$
$$\text{si } [\![e]\!]_\rho = n,\ n > 0,\ \ell \neq \texttt{null}, \text{ et } \ell \notin dom(\sigma)$$

$$\vdash_p (\texttt{dispose}(e); b, \rho)^t,\ \sigma \xrightarrow{\texttt{free}\ \ell}_t (b, \rho)^t,\ \sigma_{|dom(\sigma)-\{\ell\}} \quad \textbf{(dispose)}$$
$$\text{si } [\![e]\!]_\rho = \ell \text{ et } \ell \in dom(\sigma)$$

$$\vdash_p (x := y[e]; b, \rho)^t,\ \sigma \xrightarrow{\texttt{read}\ \ell\ n\ v}_t (b, \rho[x \mapsto v])^t,\ \sigma \quad \textbf{(get)}$$
$$\text{si } \rho(y) = \ell,\ [\![e]\!]_\rho = n,\ \sigma(\ell)[n] = v$$

$$\vdash_p (x[e_1] := e_2; b, \rho)^t,\ \sigma \xrightarrow{\texttt{write}\ \ell\ n\ v}_t (b, \rho)^t,\ \sigma \cdot [\ell, n \mapsto v] \quad \textbf{(put)}$$
$$\text{si } [\![e_1]\!]_\rho = n,\ [\![e_2]\!]_\rho = v,\ \rho(x) = \ell,$$
$$\ell \in dom(\sigma) \text{ et } \sigma(\ell)[n] \text{ défini}$$

$$\vdash_p (\texttt{while}\ e\ \texttt{do}\ b; b', \rho)^t,\ \sigma \xrightarrow{\tau}_t (b; \texttt{while}\ e\ \texttt{do}\ b; b', \rho)^t,\ \sigma \quad \textbf{(loop}_\textbf{t}\textbf{)}$$
$$\text{si } [\![e]\!]_\rho = \texttt{true}$$

$$\vdash_p (\texttt{while}\ e\ \texttt{do}\ b; b', \rho)^t,\ \sigma \xrightarrow{\tau}_t (b', \rho)^t,\ \sigma \quad \textbf{(loop}_\textbf{f}\textbf{)}$$
$$\text{si } [\![e]\!]_\rho = \texttt{false}$$

$$\vdash_p (\texttt{if}\ e\ \texttt{then}\ b_1\ \texttt{else}\ b_2; b, \rho)^t,\ \sigma \xrightarrow{\tau}_t (b_1, \rho)^t,\ \sigma \quad \textbf{(cond}_\textbf{t}\textbf{)}$$
$$\text{si } [\![e]\!]_\rho = \texttt{true}$$

$$\vdash_p (\texttt{if}\ e\ \texttt{then}\ b_1\ \texttt{else}\ b_2; b, \rho)^t,\ \sigma \xrightarrow{\tau}_t (b_2, \rho)^t,\ \sigma \quad \textbf{(cond}_\textbf{f}\textbf{)}$$
$$\text{si } [\![e]\!]_\rho = \texttt{false}$$

FIGURE 6.1 – *Sémantique opérationnelle de LUFJ : primitives séquentielles*

$$\vdash_p \phi \cdot (x := \texttt{init}(); b, \rho)^t,\ \sigma,\ \Lambda \xrightarrow{\texttt{init}\ \ell}_t \phi \cdot (b, \rho[x \mapsto \ell])^t,\ \sigma,\ \Lambda \cdot [\ell \mapsto \bullet] \qquad \textbf{(init)}$$
$$\text{si } \ell \notin dom(\Lambda)$$

$$\vdash_p \phi \cdot (\texttt{lock}\ x; b, \rho)^t,\ \sigma,\ \Lambda \xrightarrow{\texttt{lock}\ \ell}_t \phi \cdot (b, \rho)^t,\ \sigma,\ \Lambda \cdot [\ell \mapsto t] \qquad \textbf{(lock)}$$
$$\text{si } \rho(x) = \ell \text{ et } \Lambda(\ell) = \bullet$$

$$\vdash_p \phi \cdot (\texttt{unlock}\ x; b, \rho)^t,\ \sigma,\ \Lambda \xrightarrow{\texttt{unlock}\ \ell}_t \phi \cdot (b, \rho)^t,\ \sigma,\ \Lambda \cdot [\ell \mapsto \bullet] \qquad \textbf{(unlock)}$$
$$\text{si } \rho(x) = \ell \text{ et } \Lambda(\ell) = t$$

$$\vdash_p \phi \cdot (\texttt{mlock}\ b_{rl}\ e; b, \rho)^t,\ \sigma,\ \Lambda \xrightarrow{\texttt{lock}\ \ell}_t \phi \cdot (b, \rho')^t,\ \sigma,\ \Lambda \cdot [\ell \leftarrow t \mid \ell \in locks] \qquad \textbf{(multilock)}$$
$$\text{si } (locks, \rho') = (\emptyset, \rho)\ \textcircled{m}_\sigma\ b_{rl},\ \forall \ell \in locks, \Lambda(\ell) = \bullet \text{ et } [\![e]\!]_{\rho'} = true$$

$$\vdash_p \phi \cdot (y := \texttt{fork}(m, e), \rho)^{t_1}, \sigma, \Lambda \xrightarrow{\texttt{fork}\ t_2}_{t_1} \phi \cdot (b, \rho[y \mapsto t_2])^{t_1} \cdot (c, [x \mapsto v])^{t_2}, \sigma, \Lambda \qquad \textbf{(fork)}$$
$$\text{si } [\![e]\!]_\rho = v,\ m(x)\, c \in p,\ \text{et } t_2 \notin dom(\phi) \cup \{t_1\}$$

$$\vdash_p \phi \cdot (\texttt{join}\ e, \rho)^{t_1}, \sigma, \Lambda \xrightarrow{\texttt{join}\ t_2}_{t_1} \phi \cdot (b, \rho)^{t_1}, \sigma, \Lambda \qquad \textbf{(join)}$$
$$\text{si } [\![e]\!]_\rho = t_2 \text{ et } \phi(t_2) = ([], \rho_2)$$

FIGURE 6.2 – *Sémantique opérationnelle de LUFJ : primitives parallèles*

6.2.2 Règles

La relation définissant la sémantique opérationnelle de LUFJ est présentée dans les figures 6.1 et 6.2 (pages 84 et 85). C'est une relation sur un programme, deux états et un évènement (t, a) de LUFJ :

$$\vdash_p \ \phi,\ \sigma,\ \Lambda \xrightarrow{a}_t \phi',\ \sigma',\ \Lambda'$$

Dans la figure 6.1, seul le processus léger qui s'exécute est mentionné, et la table des verrous n'étant pas utilisée, elle est omise.

La règle de la sémantique

$$\vdash_p (x := e; b, \rho)^t,\ \sigma \xrightarrow{\tau}_t (b, \rho[x \mapsto v])^t,\ \sigma$$

en version complète est la suivante :

$$\vdash_p \phi \cdot (x := e; b, \rho)^t,\ \sigma,\ \Lambda \xrightarrow{\tau}_t \phi \cdot (b, \rho[x \mapsto v])^t,\ \sigma,\ \Lambda$$

Mis à part la table des verrous, les règles pour les instructions communes à AFJ sont similaires à celles d'AFJ.

La règle (**init**) permet d'initialiser un verrou. Un nom de verrou est une adresse mémoire, la variable utilisée dans cette initialisation est associé à cette adresse dans l'environnement local. La condition à respecter ici $\ell \notin dom(\Lambda)$, (ℓ étant l'adresse du verrou) est que l'adresse n'ait jamais servie en tant que verrou.

La règle (**lock**) modélise le verrouillage d'un verrou. Nous imposons que le verrou ne soit pas déjà pris. Nos verrous ne sont donc pas réentrants. La règle (**unlock**) modélise le déverrouillage d'un verrou. Le verrou doit être verrouillé. Classiquement nous imposons que le verrouillage et le déverrouillage doivent être fait par le même processus léger.

La règle (**multilock**) permet de lire et de verrouiller plusieurs verrous en un pas. Le bloc d'instructions b_{rl} contient des instructions de lecture et de prise de verrous. L'opérateur ⓜ (défini ci-dessous) appliqué à cette liste modifie l'environnement local pour prendre en compte ces lectures. Cette opération produit également un ensemble de verrous. La table des verrous est modifiée pour chacun de ces verrous. Enfin une condition est présente : elle doit être vérifiée.

Définition 6.1 $\text{ⓜ} : (2^{\mathbb{L}} \times (\mathcal{X} \rightharpoonup \mathcal{V})) \to instr_{rl} \to (2^{\mathbb{L}} \times (\mathcal{X} \rightharpoonup \mathcal{V}))$

L'opération ⓜ prend en argument une paire constituée d'un ensemble de verrous et un environnement local, et une instruction de $instr_{rl}$ (soit une lecture, soit une prise de verrou), et applique le second sur le premier. Dans le cas d'une lecture, l'environnement local est modifié. Dans le cas d'une prise de verrou, celui-ci est ajouté à l'ensemble s'il n'y est pas déjà présent.

$$\frac{[\![e]\!] = n \ \wedge\ \rho(y) = \ell \ \wedge\ \sigma(\ell)[n] = v}{(locks, \rho)\text{ⓜ}_\sigma(z := y[e]) \ =\ (locks, \rho[z \mapsto v])} \qquad \frac{[\![x]\!] = \ell}{(locks, \rho)\text{ⓜ}_\sigma(\texttt{lock}\ x) \ =\ (locks \cup \{\ell\}, \rho)}$$

Par abus de notation, nous notons également ⓜ l'opérateur de réduction associé.

6.2.3 Choix de conception

Afin de simplifier cette sémantique, et surtout les preuves associées à la correction de la compilation présentée aux chapitres suivants, nous avons exclu l'instruction de désallocation de verrou. Il est bien sûr possible de rajouter cette instruction. Comme nous allons le voir dans la section 6.3 pour les pointeurs, rajouter cette possibilité aurait nécessité de faire la extension pour les verrous similaire à celle que nous faisons pour les pointeurs. Il nous a semblé que ces détails sont orthogonaux au principe de compilation et nous avons opté pour une sémantique plus simple de ce point de vue.

Concernant l'instruction `mlock`, nous aurions pu avoir une approche de plus bas-niveau, en ajoutant une instruction `trylock` à notre langage, avec les règles de sémantique :

$$(\texttt{trylock}\ y\ x; b,\ \rho),\ \sigma,\ \Lambda \xrightarrow{\texttt{lock}\ \ell}_t (b,\ \rho[y \mapsto \mathit{true}]),\ \sigma,\ \Lambda \cdot [\ell \mapsto \texttt{t}] \qquad (\mathbf{trylock_t})$$
$$\text{si } \rho(x) = \ell \text{ et } \Lambda(\ell) = \bullet$$

$$(\texttt{trylock}\ y\ x; b,\ \rho),\ \sigma,\ \Lambda \xrightarrow{\tau}_t (b,\ \rho[y \mapsto \mathit{false}]),\ \sigma,\ \Lambda \qquad (\mathbf{trylock_f})$$
$$\text{si } \rho(x) = \ell \text{ et } \Lambda(\ell) \neq \bullet$$

Une instruction $\texttt{mlock}\ \{\ldots; x_0^i := y_0^i[e_0^i]; \ldots; x_{k_i}^i := y_{k_i}^i[e_{k_i}^i]; \texttt{lock}\ z_i; \ldots\}\ e$ pourrait ainsi être remplacée par :

all_locked := **false**;
while !*all_locked* **do** {
  ...
    x_0^i:=$y_0^i[e_0^i]$;
    ...
    $x_{k_i}^i$:=$y_{k_i}^i[e_{k_i}^i]$;
    $locked_i$ = **trylock**(z_i)
    if $locked_i$ || z_0=z_i || ... || z_{i-1}=z_i
    then {
      x_0^{i+1}:=$y_0^{i+1}[e_0^{i+1}]$;
      ...
    } **else** {
      unlock(z_0); ...; **unlock**(z_{i-1}) }
    }
  ...
  if *e*
  then *all_locked*:=**true**
  else { **unlock**(z_0); ...; **unlock**(z_m) }
}

Cette implantation de plus bas niveau rend beaucoup plus complexe le code à traiter pour la correction de la compilation de AFJ vers LUFJ. Le langage avec l'instruction

`mlock` peut ainsi être considéré comme une étape intermédiaire, l'étape suivante serait de remplacer les appels à `mlock` par le code ci-dessus.

Dans cette hypothèse, il faudrait s'interroger sur l'introduction ou non des signaux. Ces derniers permettent d'éviter l'idiome de codage « attente active ». Toutefois au niveau des traces, si on retient une solution similaire à celle qui a cours dans les bibliothèques de *PThreads*, il ne nous a pas semblé que ceci faisait une grande différence, tout en introduisant des catégories syntaxiques et des règles supplémentaires. Là encore dans un souci de simplicité, et parce que ceci semble orthogonal à la correction du principe de compilation, nous écarterions cette solution et opterions pour ne garder que des verrous et utiliser des boucles d'attente active dans le code. Dans une implantation pratique de la compilation ceci pourrait être remplacé par des signaux, et la correction pourrait être établie au prix de détails plus nombreux mais à notre avis sans changements fondamentaux.

6.3 Prise en compte des pointeurs pendants

Dans les chapitres suivants, nous allons nous intéresser à la préservation sémantique d'une passe de transformation de AFJ vers LUFJ. Un point important pour raisonner sur ce type de transformation, est de pouvoir mettre en relation les états mémoires de la sémantique cible et de la sémantique source. La capture de pointeurs pendants, c'est-à-dire des adresses mémoires non allouées mais présentes comme données dans la mémoire, pose alors un gros problème.

Puisque notre transformation vise essentiellement à changer l'utilisation de sections atomiques en utilisation de verrous, on peut imaginer que nous soyons dans la situation suivante :

— dans l'état cible LUFJ, une variable x contient une adresse mémoire ℓ_0 qui elle-même contient une autre adresse ℓ pendante,
— dans l'état source AFJ équivalent, une variable x contient une adresse mémoire ℓ'_0 qui elle-même contient une autre adresse ℓ' pendante,
— l'instruction à exécuter aussi bien dans le programme cible que le programme source est `z:=allocate(1)`.

L'allocation mémoire étant indéterministe, l'exécution de cette instruction peut très bien, par exemple, allouer à nouveau l'adresse ℓ pour le programme LUFJ, et allouer une adresse ℓ'' différente de ℓ' pour le programme AFJ (ou inversement). On voit bien alors que les états mémoire résultants ne peuvent plus être considérés comme équivalents.

Une solution est de marquer les pointeurs qui deviennent pendants. Une façon de le faire est de considérer des adresses que nous appellerons logiques, constituées d'une adresse physique a et d'un compteur d'allocation n indiquant le nombre de fois que cette adresse a été allouée. Pour l'écriture et la lecture, on considère l'adresse logique. Il est ainsi invalide d'accéder à une adresse mémoire en utilisant une valeur potentiellement capturée fortuitement Dans l'exemple ci-dessus, si on considère ces adresses

logiques, même si la même adresse physique est à nouveau allouée, le compteur de la seconde allocation sera différent de la première, il est donc impossible d'allouer à nouveau la même adresse logique ℓ. Les états seraient donc équivalents.

Un dernier point à examiner est celui de la comparaison d'adresses, qui est la seule opération prédéfinie autorisée sur les adresses. La comparaison peut se faire uniquement sur la partie physique de l'adresse, mais il est indispensable que l'évaluation d'une comparaison ne puisse se faire que sur des adresses allouées et `null`. En effet dans le cas contraire, le code suivant pourrait diverger entre la cible et la source :

if $x{=}y$ **then** $c{:=}c{+}1$ **else** $c{:=}c{-}1$

dans le cas où `x` et `y` contiennent une adresse logique ayant la même adresse physique, mais un compteur différent.

Pour modéliser ceci, une adresse ℓ est désormais un couple (a, n), et une fonction $\mathcal{A}$ qui stocke le nombre d'allocations pour chaque adresse physique, est ajoutée aux états LUFJ.

La plupart des règles sont presque inchangées : le changement de nature des adresses mémoire n'affecte par les règles, et que la fonction $\mathcal{A}$ n'est pas utilisée. Par contre l'évaluation des expressions est modifiée. Pour que la comparaison d'adresses ne puissent se faire qu'avec des adresses allouées, l'évaluation doit pouvoir accéder l'état. L'évaluation devient donc $[\![e]\!]_{\rho,\sigma}$, pour toute les règles. Enfin la règle d'allocation est modifiée de façon plus marquée et devient :

$$\vdash_p (x := \texttt{allocate}(e); b, \rho)^t,\ \sigma, \mathcal{A}$$
$$\xrightarrow{\texttt{alloc}\ \ell\ n}_t \qquad \textbf{(alloc')}$$
$$(b, \rho[x \mapsto \ell])^t,\ \sigma \cdot [\ell \mapsto [0]^n], \mathcal{A} \cdot [a \mapsto \mathcal{A}(a) + 1]$$
$$\text{si } [\![e]\!]_{\rho,\sigma} = n,\ n > 0,\ \ \forall n, (a, n) \notin dom(\sigma),\ \ell = (a, \mathcal{A}(a) + 1)$$

Une propriété structurelle du code exécuté

Pour l'étude de la correction de la compilation de AFJ vers LUFJ, il est nécessaire de raisonner sur la structure du code des programmes LUFJ, au cours de l'exécution. La sémantique de la boucle et de la conditionnelle est telle que le code attaché à un processus léger au cours de l'exécution n'est pas simplement un suffixe du code principal du programme ou du corps d'une méthode.

Nous définissons tout d'abord un prédicat de caractérisation du code attaché aux processus légers atteignable au cours d'une exécution, puis nous énonçons et donnons les grandes lignes de la preuve que le code résultant d'une exécution suit effectivement cette caractérisation.

Le prédicat *executedCodeOf p b* indique que le bloc b est du code qui peut être associé à un processus léger lors de l'exécution du programme p. Ce prédicat est défini par le système d'inférence de la figure 6.3 où $main(p)$ est le bloc principal du programme p et $methods(p)$ la liste des méthodes du programme p.

$$\frac{main(p) = done \mathbin{+\!\!+} code}{executedCodeOf\ p\ code} \tag{6.1}$$

$$\frac{\begin{array}{l} main(p) = done \cdot (\texttt{while}\ e\ \texttt{do}\ boby) \mathbin{+\!\!+} suffix \\ executedCodeOf\ (methods(p)\ body)\ code \end{array}}{executedCodeOf\ p\ code \cdot (\texttt{while}\ e\ \texttt{do}\ boby) \mathbin{+\!\!+} suffix} \tag{6.2}$$

$$\frac{\begin{array}{l} main(p) = done \cdot (\texttt{if}\ e\ \texttt{then}\ b_{then}\ \texttt{else}\ b_{else}) \mathbin{+\!\!+} suffix \\ executedCodeOf\ (methods(p)\ b_{then})\ code \end{array}}{executedCodeOf\ p\ code \mathbin{+\!\!+} suffix} \tag{6.3}$$

$$\frac{\begin{array}{l} main(p) = done \cdot (\texttt{if}\ e\ \texttt{then}\ b_{then}\ \texttt{else}\ b_{else}) \mathbin{+\!\!+} suffix \\ executedCodeOf\ (methods(p)\ b_{else})\ code \end{array}}{executedCodeOf\ p\ code \mathbin{+\!\!+} suffix} \tag{6.4}$$

$$\frac{m(x)body \in methods(p) \quad executedCodeOf\ (methods(p)\ body)\ code}{executedCodeOf\ p\ code} \tag{6.5}$$

FIGURE 6.3 – *Caractérisation du code au cours de l'exécution*

Par exemple la règle 6.1 indique que le bloc *code* peut être associé à un processus léger lors de l'exécution du programme p, si le bloc d'instruction complet du programme p peut être décomposé en deux blocs *done* et *code*, *done* représentant le bloc d'instructions déjà exécutées.

Lemme 6.1 *Soit p un programme, s une trace et $(\phi, \sigma, \Lambda, \mathcal{A})$ un état tels que $p \xhookrightarrow{s} (\phi, \sigma, \Lambda, \mathcal{A})$. Pour tout $(code, \rho)^t \in \phi$, on a executedCodeOf p code.*

Démonstration. La preuve se fait par induction sur la trace s. Le cas de base est immédiat par (6.1). Soit $s = s' \cdot (t, a)$. On raisonne ensuite par cas sur la réduction qui produit le dernier évènement t. Tout d'abord notons que pour tout identifiant de processus léger $t' \neq t$, le code ne change pas, sauf dans le cas où l'action est une création de processus léger auquel cas si $t' = t_0$ où t_0 est l'identifiant du processus léger créé, alors le code est le corps d'une méthode et on conclut par (6.5) et (6.1).

Reste à considérer les cas où $t' = t'$. Nous considérons alors tous les cas possibles de réduction (15 cas). Pour chacun des cas nous raisonnons par induction sur le jugement *executedCodeOf* obtenu par application de l'hypothèse d'induction (sur la trace). La structure de la preuve est alors identique pour toutes les instructions sauf la boucle, la conditionnelle et la création de processus légers. Il s'agit de reconstruire, souvent

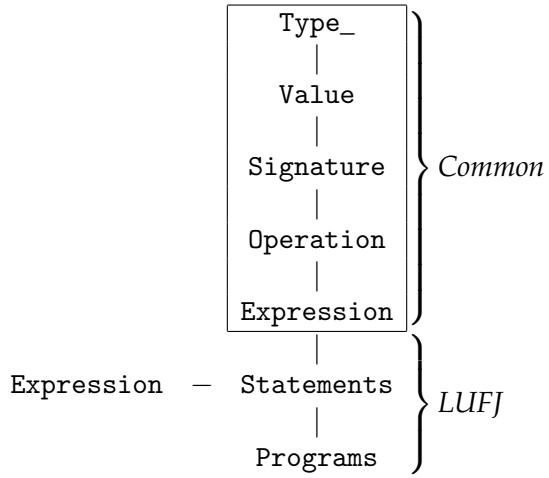


FIGURE 6.4 – *Organisation des bibliothèques Coq pour LUFJ*

avec la même règle que celle appliquée pour obtenir le jugement depuis l'hypothèse d'induction, avec parfois en plus la règle (6.1). □

6.4 Formalisation en Coq

6.4.1 Organisation et choix de formalisation

Elle est similaire à l'organisation décrite pour AFJ. Néanmoins, cas le cas de la sémantique tenant compte des pointeurs pendants, l'évaluation modifiée des expressions et du nombre d'allocations dans l'état introduisent un peu de modifications. L'organisation générale des bibliothèques pour LUFJ est résumée sur la figure 6.4.

Ainsi le module des types de LUFJ, qui comme pour AFJ, prend en argument un module de type DecidableInfiniteSet n'est pas instancié avec un module Ad de ce type (le type abstrait des adresses physique) mais avec un module Na qui est construit à partir de Ad par le foncteur NumAlloc et qui comme son nom l'indique rajoute aux adresses physiques un entier qui compte le nombre d'allocations pour cette adresse physique. Tous les foncteurs qui suivent sont instanciées avec Na au lieu d'être instanciés par Ad.

Le foncteur des expressions prend quant à lui en argument un foncteur d'opération prédéfinies. Le type de ce foncteur (nommé T) est défini dans une bibliothèque Common.Operation qui est commune à tous les langages exposés dans ce mémoire. Ce foncteur a le type suivant :

```
Module Type T (Import Ad: DecidableInfiniteSet)(Import Ty : Type_.TYPE Ad)
              (Import Signature : Signature.T Ad Ty).
  Parameter operation : Set.
  Parameter eq_dec : ∀(o o' : operation), { o = o' } + { ~o = o'}.
  Parameter opSignature : operation →Signature.t.
  Parameter state : Type.
  Parameter opDenote: state →∀(op:operation), Signature.typeOf(opSignature op).
End T.
```

On peut noter la présence du type state dans cette signature et le fait que la fonction opDenote, qui fait correspondre à une opération la fonction Coq qui implante sa sémantique, prend en argument un état. C'est indispensable pour que l'évaluation des expressions puissent bloquer lors de l'utilisation de pointeurs pendants dans des expressions de comparaison.

Dans le chapitre 5, nous ne nous somme pas soucié du problème des pointeurs pendants. La bibliothèque Operation contient un foncteur Simple qui implante de façon paramétrée un jeu simple d'opérations qui peut être utilisé quand on ne s'occupe pas du problème des pointeurs pendants. Dans ce module, le type state ne sert à rien : est simplement défini comme étant unit le type prédéfini Coq ne contenant qu'une seule valeur tt.

La bibliothèque LUFJ.Operation contient quant à elle un foncteur T qui prend des arguments supplémentaires par rapport au type de module T de Common.Operation. Essentiellement ces modules supplémentaires sont un module qui définit le type du tas et les opérations associées, ainsi que ses dépendances. Pour décrire le type de ce module il faut également pouvoir parler des valeurs, enfin comme nous souhaitons indiquer que ce foncteur respecte la signature Common.Operation.T, il faut également pouvoir parler d'un module de signatures tel que présenté dans la section 5.4 (page 74). Le module LUFJ.Operation.T peut alors proposer la fonction opDenote qui pour le cas de l'égalité entre deux adresses pourra être définie comme suit :

Nom / Description	Coq	Fichier
Module d'opérations	T	`LUFJ/Operation.v`
Instructions	t	`LUFJ/Statements.v`
Méthodes	method	`LUFJ/Programs.v`
Programmes	t	`LUFJ/Programs.v`
Actions et évènements	action et t	`LUFJ/Event.v`
Module de définition du tas	T	`LUFJ/Heap.v`
Opération ⓜ	mlock_binary	`LUFJ/Programs.v`
Relation $\vdash_p \ \Sigma \xrightarrow{a}_t \Sigma'$	sem	`LUFJ/Programs.v`
Predicat *executedCodeOf*	executedCodeOf	`LUFJ/Programs_Basics.v`
Lemme 6.1	code_characterisation	`LUFJ/Programs_Basics.v`

FIGURE 6.5 – *Table de correspondance avec les termes Coq*

```
| Equal Allocation ⇒
    fun x y ⇒
      let checkx :=
          if (He.in_domain_dec st x)
          then true
          else if (Na.eq_dec x null)
                then true
                else false in
      let checky :=
          if (He.in_domain_dec st y)
          then true
          else if (Na.eq_dec y null)
                then true
                else false in
      if (andb checkx checky)
      then Ok(if Na.eq_dec x y then true else false)
      else Error "Invalid pointer"
```

où He.in_domain_dec st x décide si l'adresse *logique* x est bien dans l'état (ici seulement le tas) st.

6.4.2 Correspondance

Le développement en Coq est disponible l'adresse suivante :

`http://traclifo.univ-orleans.fr/PaPDAS/wiki/TransactionsInCoq`

La correspondance entre les définitions et propositions introduites dans ce chapitre et leurs contreparties Coq est résumée dans le tableau de la figure 6.5.

Compilation de AFJ vers LUFJ

7

Sommaire

Nous avons donc maintenant deux langages, AFJ le langage source et LUFJ le langage cible. Nous proposons dans ce chapitre un processus de compilation de AFJ vers LUFJ. Des structures de données et des opérations les manipulant sont nécessaires et forment un noyau de support à l'exécution : dans un premier temps nous les présentons. La fonction de compilation peut alors être définie.

7.1 Structures de données

L'objectif du processus de compilation est de remplacer les sections atomiques par du code offrant la même sémantique en ce qui concerne les synchronisations, mais en utilisant des verrous à la place. En tant que première version de compilation qui change la nature des synchronisations, nous avons opté pour un principe de compilation qui explicite la structure de bulle de la sémantique opérationnelle de AFJ sous forme d'une structure de données, et les opérations la manipulant sous forme de code LUFJ. Ce n'est pas la transformation la plus efficace qui soit, mais la propriété de bonne formation ($\mathbf{wf}_3$) limite pour le moment l'efficacité des implantations alternatives qui pourraient être imaginées.

Cette structure de données est naturellement appelée bulle et est schématisée dans la figure 7.1. Nous penserons à cette structure comme à un enregistrement même si dans la syntaxe de LUFJ ceci est réalisé par un pointeur et des accès à l'aide de décalages. Elle est composée de deux pointeurs (previous, next) vers les bulles de niveau immédiatement supérieur et inférieur (si elles existent). Un verrou lockB est présent pour contrôler les modifications sur ces deux pointeurs. Un pointeur (first) permet d'accéder au premier élément d'une liste d'éléments représentant les processus légers

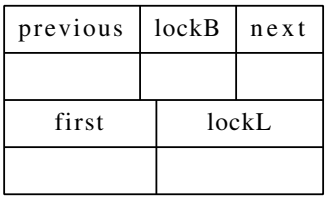


FIGURE 7.1 – *Structure de données* bubble *représentant une bulle*

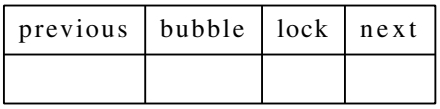


FIGURE 7.2 – *Structure de données* info *représentant un processus léger*

présents dans cette bulle. Un second verrou (lockL) permet de contrôler les modifications sur le pointeur de tête de liste. La liste est doublement chaînée et forme un cycle.

Les éléments de cette liste peuvent également être vus comme des enregistrements. Un tel élément info est schématisé figure 7.2. L'enregistrement contient un pointeur vers la bulle où le processus est présent (bubble). Un verrou (lock) permet de contrôler les modifications apportées à cette structure. Deux pointeurs (previous, next) vers des structures de même type réalisent le double chaînage. La liste forme un cycle, donc l'élément précédant le premier est le dernier et inversement.

Chaque processus léger n'a pas d'accès direct à la structure de bulle. Par contre chacun a dans son environnement local, un pointeur vers l'élément info le représentant. Celui-ci contenant un pointeur vers la structure de bulle, l'accès est indirect. Les processus légers sont amenés à ouvrir de nouvelles sections atomiques, c'est-à-dire créer de nouvelles bulles, et à fermer des sections atomiques, c'est-à-dire supprimer des bulles. C'est le processus responsable de cette création ou suppression qui sera amené à transférer les processus d'une structure de bulle à une autre. Un accès direct à la structure de bulle depuis *l'environnement local* d'un processus léger n'aurait pas permis ce transfert d'information de processus d'une bulle à une autre tout en conservant la cohérence entre l'information locale d'appartenance à une bulle et l'appartenance effective. L'accès d'un processus léger à sa structure info est lui immuable : cette structure est créée au moment de la création du processus et elle n'est détruite qu'à la terminaison du processus. Tout au long de la vie du processus, la variable locale qui contient l'adresse de cette structure ne change pas.

Exemple *Considérons une bulle de la sémantique opérationnelle du chapitre 5 :*

$$(\!| (c_1, \rho_1)^{t_1} \cdot (c_2, \rho_2)^{t_2} \cdot (c_3, \rho_3)^{t_3}; (\!| (c_4, \rho_4)^{t_4}; \circ |\!)^{s', t_4}_{C} |\!)^{s, t_4}_{\square}$$

La structure de données en mémoire encodant une telle bulle est présentée à la figure 7.3.

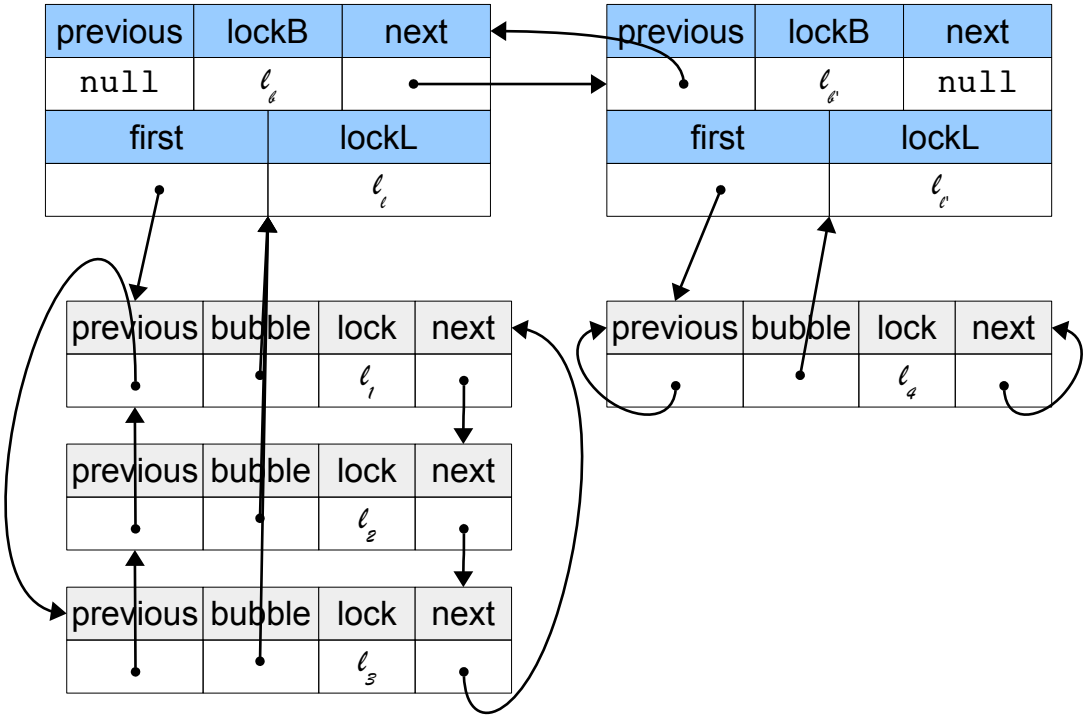


FIGURE 7.3 – *Exemple de structure de bulle en mémoire*

$$
\begin{array}{rcl}
\mathcal{C}(\overline{mth}\ b) & = & \overline{\mathcal{C}(mth)}\{\#initialize; \mathcal{C}(b)\} \\
\mathcal{C}(m(x)\ b) & = & m(arg)\{\#forkCB(x); \mathcal{C}(b); \#forkCE\} \\
\mathcal{C}(c; b) & = & \mathcal{C}(c); \mathcal{C}(b) \\
\mathcal{C}([]) & = & [] \\
\mathcal{C}(x := \texttt{fork}(m, e)) & = & \#forkPB(e); \mathcal{C}(x) := \texttt{fork}(m, arg) \\
\mathcal{C}(\texttt{atomic}\ \{c\}) & = & \#open; \mathcal{C}(c); \#close \\
\mathcal{C}(\texttt{if}\ e\ \texttt{then}\ b_1\ \texttt{else}\ b_2) & = & \texttt{if}\ e\ \texttt{then}\ \mathcal{C}(b_1)\ \texttt{else}\ \mathcal{C}(b_2) \\
\mathcal{C}(\texttt{while}\ e\ \texttt{do}\ b) & = & \texttt{while}\ e\ \texttt{do}\ \mathcal{C}(b) \\
\mathcal{C}(c) & = & c \qquad \text{pour tous les autres cas}
\end{array}
$$

FIGURE 7.4 – *Fonction de compilation*

7.2 Fonction de compilation

Intéressons nous maintenant à l'utilisation de ces structures, en détaillant le processus de compilation. Nous considérons une fonction $\mathcal{C}$ des programmes AFJ vers les programmes LUFJ. Cette fonction de compilation est récursive et nous la présentons surchargée : elle peut être appliquée à des programmes, des définitions de méthodes, des listes de commandes, des commandes. Cette fonction est définie figure 7.4. Les cas de compilation les plus intéressants et que nous allons détailler concernent la transformation des sections atomiques et la création de processus légers. Ces transformations consistent à ajouter du code pour la manipulation des structures de données de bulles. Du fait de ces ajouts, on peut distinguer deux types de code : le code classique directement issu du programme source, et le code instrumenté. Ce code instrumenté est indiqué en bleu dans la figure 7.4. Nous avons choisi d'utiliser un style de macro-commandes. Certaines de ces macro-commandes sont constantes, d'autres sont paramétrées. Par convention les macro-commandes commencent par le symbole #.

Initialisation des structures de données Le préambule du programme compilé (*#initialize*) met en place la première bulle. Le code détaillé est présenté figure 7.5. Il consiste à créer la première bulle et à y ajouter le premier élément info représentant le processus léger initial. Ce code fait appel à d'autres macro-commandes dont les noms sont explicites et qui sont détaillées en section 7.A. Comme déjà évoqué plus haut, la syntaxe de LUFJ n'a pas d'enregistrement, mais une notation de tableaux. Pour faciliter la lecture du code, nous employons des constantes symboliques pour les indices de tableaux : ainsi nous notons info[F_BUBBLE] plutôt que info[1]. Les constantes symboliques sont toutes construites sur le même principe : F_ suivi du nom d'un champ présenté dans les figures 7.1 et 7.2.

Sections atomiques Intéressons nous maintenant à la compilation des sections atomiques. Cela consiste à exécuter le code compilé de la section précédé d'une suite d'instructions d'ouverture de section (*#open*) et suivi d'une suite d'instructions liées à

```
0  #INITIALIZE ≡
1      #newInfo null info ;
2      #newBubble nBubble ;
3      info[F_BUBBLE] := nBubble;
4      nBubble[F_FIRST] := info
```

FIGURE 7.5 – *Code ajouté pour l'initialisation des programmes LUFI*

```
0  #OPEN ≡
1    mlock( locki := info[F_LOCK]; lock locki;
2           bubble := info[F_BUBBLE]; lockb := bubble [F_LOCK_B]; lock lockb;
3           next := bubble[F_NEXT] )
4         ( next = null );
5    #newBubble nBubble ;
6    bubble := info[F_BUBBLE];
7    bubble[F_NEXT] := nBubble;
8    nBubble[F_PREVIOUS] := bubble;
9    #remove info ;
10   nBubble[F_FIRST] := info;
11   info[F_NEXT] := info;
12   info[F_PREVIOUS] := info;
13   unlock(lockB)
```

FIGURE 7.6 – *Code ajouté pour l'ouverture de sections atomiques*

la fermeture de la section (*#close*). Ces deux macro-commandes sont détaillées respectivement dans les figures 7.6 et 7.7.

Pour ouvrir une section, il est nécessaire qu'aucune sous-section n'existe. Le processus léger qui ouvre la section s'assure donc que le champ next de sa bulle soit null. Pour ce faire il doit tout d'abord verrouiller sont propre verrou puis récupérer l'adresse de sa structure de bulle pour ensuite pouvoir prendre le verrou de bulle. Une fois ces deux verrous pris, il peut tester le champ next. Si l'ouverture n'est pas possible, ou si la prise successive des deux verrous n'est pas possible, les verrous sont relâchés, puis le processus recommence la procédure complète de test. Cela permet d'éviter la monopolisation du verrou par le processus, et nécessite l'emploi de l'instruction `mlock`. Si l'ouverture est possible, le verrou de bulle est gardé jusqu'à la fin de cette phase. Le processus crée ensuite la nouvelle bulle, qui est liée à la précédente, et ajoute son info à cette nouvelle structure. Lors de ce déplacement il est nécessaire de prendre (puis relâcher) les verrous des infos qui sont modifiés (c'est-à-dire l'élément déplacé et ses voisins immédiats). Il relâche ensuite le verrou de la bulle d'origine.

De manière similaire à l'ouverture, une fermeture n'est possible que si aucune sous section n'existe. Le processus léger qui souhaite fermer sa section s'assure que le champ next de sa bulle soit null. Des processus légers créés dans cette section peuvent toujours être en train de s'exécuter dans ce niveau. Ils doivent pouvoir continuer leur exécution malgré la fin de la section englobante. Cela correspond à l'échappement de processus. Les processus de ce niveau sont donc déplacés dans la bulle de niveau

```
0  #CLOSE ≡
1    mlock[ locki  := info[F_LOCK]); lock locki;
2             bubble  := info[F_BUBBLE]; lockb  := bubble[F_LOCK_B]; lock lockb;
3             next  := bubble[F_NEXT] ]
4          ( next == null );
5    lockL  := bubble[F_LOCK_L];
6    lock (lockL);
7    first  := bubble[F_FIRST];
8    previousBubble  := bubble[F_PREVIOUS];
9    while first != null do
10   [
11     cell  := first;
12     #removeFirst cell ;
13     unlock(lockL);
14      #addFirst cell ;
15     lock (lockL);
16     first  := bubble[F_FIRST]
17   ];
18   unlock(lockL);
19
20   lockBP  := previousBubble[F_LOCK_B];
21   lock (lockBP);
22   previousBubble[F_NEXT]  := null;
23   unlock(lockBP);
24   dispose(bubble)
```

FIGURE 7.7 – *Code ajouté pour la fermeture de sections atomiques*

```
0  #forkPB e ≡
1    arg  := allocate(2);
2    parentlock  := info[F_LOCK];
3    mlock( lock parentlock;
4           next  := info[F_NEXT]; lockN  := next[F_LOCK]; lock lockN )
5           True;
6
7    bubble  := info[F_BUBBLE];
8    #newInfo bubble childInfo ;
9
10   childInfo[F_NEXT]  := next;
11   info[F_NEXT]  := childInfo;
12   next[F_PREVIOUS]  := childInfo;
13   childInfo[F_PREVIOUS]  := info
14
15   #unlockDifferent lockN parentlock;
16   #unlockDifferent parentlock lockN ;
17   arg[0]  := childInfo;
18   arg[1]  := e
```

FIGURE 7.8 – *Code ajouté pour la création de processus léger*

```
0  #forkCB x ≡
1    info  := arg[0];
2    x  := arg[1];
3    dispose(arg)
```

```
0  #forkCE ≡
1    #remove info ;
2    dispose(info)
```

FIGURE 7.9 – *Code ajouté au début et à la fin de chaque méthode*

immédiatement inférieur, en ayant pris soin de prendre les verrous des infos concernés. La bulle est ensuite détruite.

Création de processus léger Le dernier cas intéressant de compilation à aborder est celui de la création de processus légers. La première partie liée à la compilation de création de processus léger consiste à transformer le code du processus parent (figure 7.8). Chaque processus léger doit avoir dans son environnement local, une référence vers la structure info le représentant. Il est donc nécessaire de le renseigner lors de la création d'un processus. Le processus parent créé donc la structure info de son processus fils. Cette structure est transmise via l'argument de la méthode, un tableau arg de taille deux, créé par le processus parent et qui sera lu ensuite par le nouveau processus. La première case contient la structure info du nouveau processus, et la se-

conde l'argument original de la méthode. Le processus parent ajoute cette nouvelle structure info à sa bulle.

Examinons maintenant la seconde partie liée à la création de processus léger, c'est-à-dire l'exécution de la méthode donnée en paramètre dans le `fork` (figure 7.9). Ces macro-commandes sont utilisées respectivement au début et à la fin du processus créé. Cela débute par la récupération des informations transmises par le processus parent via arg. Ensuite le processus léger exécute le code d'origine. Enfin il se retire de la liste des processus de la bulle. Il est à noter que lors d'une suppression, le processus doit se verrouiller lui-même, ses deux voisins et éventuellement s'il est en première position dans la liste le verrou associé au pointeur de tête de liste.

Discussion Nous discutons informellement de l'entrelacement possible du code de ces macro-commandes. La phase d'initialisation marque le début du programme, aucun autre processus léger n'est lancé, il n'y a donc pas d'entrelacement possible avec d'autres portions de code, instrumenté ou non.

Les phases d'ouverture et de fermeture de sections limitent leurs entrelacements grâce au verrou de bulle lockB qui est pris au début de chacune des phases. Les deux parties respectives exécutées après la prise du verrou sont en exclusion mutuelle.

La phase d'ouverture de section et de création de processus léger peuvent s'entrelacer, mais cela a des conséquences différentes suivant la position des processus légers dans la structure chaîné de bulles. Si le processus léger qui veut créer un nouveau processus et celui voulant effectuer l'ouverture de la section ne sont pas au même niveau, ils modifient chacun une liste d'info différente, il n'y a donc pas de problème d'accès conflictuel. Si ces deux processus sont dans la même bulle, ils modifient la même liste, avec les fonctions remove et add. Néanmoins chacune de ces méthodes prend le verrou des éléments info qui sont modifiés. Il n'y a donc pas non plus de conflits. Les verrous sont soit tous pris soit aucun n'est pris grace à l'utilisation de `mlock` : il n'y a donc pas non plus de problème d'interblocage.

L'entrelacement de la phase de fermeture de section et de création de processus léger est également particulière. Durant la phase de fermeture les processus légers sont déplacés vers la bulle précédente. De manière similaire à l'ouverture, les verrous pris sur les éléments info modifiés empêchent les conflits entre ces deux phases.

7.A Macro-commandes LUFJ complémentaires

Nous présentons ici le code tel qu'il a été écrit en utilisant une formalisation de la syntaxe de LUFJ en Coq, des notations Coq pour faciliter l'écriture des termes Coq représentant des instructions LUFJ et enfin en utilisant un style pour le paquet `listings` de LaTeX afin que le rendu soit plus agréable à lire. Comme précédemment les identifiants qui commencent par # sont considérés comme des macro-commandes.

newInfo

```
0  #newInfo bubble ret ≡
1    ret  := allocate(4);
2    ret[F_PREVIOUS]  := ret;
3    lock  := init();
4    ret[F_LOCK]  := lock;
5    ret[F_NEXT]  := ret;
6    ret[F_BUBBLE]  := bubble
```

newBubble

```
0  #newBubble ret ≡
1    ret  := allocate(7);
2    ret[F_PREVIOUS]  := null;
3    ret[F_NEXT]  := null;
4    lockB  := init();
5    lockL  := init();
6    ret[F_LOCK_B]  := lockB;
7    ret[F_LOCK_L]  := lockL;
8    ret[F_FIRST]  := null
```

unlockDifferent

```
0  #unlockDifferent (lock1 lock2 : var) ≡
1    if (lock1 == lock2)
2       then [ ]
3    else [ unlock(lock1) ]
```

removeFirst

```
0  #removeFirst ≡
1    mlock( lockc  := cell [F_LOCK]; lock lockc;
2           previous  := cell[F_PREVIOUS]; lockp  := previous[F_LOCK]; lock lockp;
3           next  := cell [F_NEXT]; lockn  := next [F_NEXT]; lock lockn )
4           True;
5    bubble  := cell[F_BUBBLE];
6    #unlockDifferent lockP lock;
7    #unlockDifferent lockN lockP;
8    if (cell == previous)
9    then [ bubble[F_FIRST]  := null ]
10   else [ bubble[F_FIRST]  := next ];
11   unlock(lock);
```

remove

```
0  remove (cell : var) ≡
1    mlock( lock  := cell [F_LOCK]; lock lock;
2           previous  := cell [F_PREVIOUS]; lockP  := previous [F_LOCK]; lock lockP;
3           next  := cell [F_NEXT]; lockN  := next [F_LOCK]; lock lockN )
4         True;
5    previous[F_NEXT]  := next;
6    next[F_PREVIOUS]  := previous;
7    bubble  := cell[F_BUBBLE];
8    lockL  := bubble[F_LOCK_L];
9    lock (lockL);
10   #unlockDifferent lockP lock;
11   #unlockDifferent lockN lockP;
12   first  := bubble[F_FIRST];
13   if (first == cell)
14   then [ if (cell == previous)
15          then [ bubble[F_FIRST]  := null ]
16          else [ bubble[F_FIRST]  := next ]
17   else [ ];
18   unlock(lockL);
19   unlock(lock);
```

addFirst

```
0  #addFirst cell ≡
1    bubble  := cell[F_BUBBLE];
2    pBubble  := bubble[F_PREVIOUS];
3    lockL  := pBubble[F_LOCK_L];
4    mlock( lock lockL;
5           cFirst :: =pBubble [F_FIRST]; lockF  := cFirst [F_LOCK]; lock lockF )
6         True;
7    if (first == null) then
8    [
9      bubble[F_FIRST]  := cell;
10     cell[F_NEXT]  := cell;
11     cell[F_PREVIOUS]  := cell;
12   ] else [
13     first[F_PREVIOUS]  := cell;
14     cell[F_NEXT]  := first;
15     last[F_NEXT]  := cell;
16     cell[F_PREVIOUS]  := last;
17     bubble[F_FIRST]  := cell;
18   ];
19   unlock(lockF);
20   unlock(lockL)
```

8 Vérification d'une compilation pour un langage séquentiel

Sommaire

Dans ce chapitre, nous nous intéressons à l'équivalence entre un programme source et sa version compilée pour un langage séquentiel. Le langage source et cible sont identiques. La passe de compilation est une transformation source à source qui introduit une structure de données chaînée qui stocke l'ensemble des adresses allouées avec la taille allouée à chaque adresse, et les opérations permettant que cette structure reflète l'état d'allocation mémoire tout au long de l'exécution.

Nous ne considérons ainsi pas encore les langages *AFJ* et *LUFJ*, mais un sous-ensemble d'instructions communes à ces deux langages. Nous utilisons donc un langage séquentiel sans création de sections atomiques ou de verrous. En ne considérant qu'un noyau du langage commun à *AFJ* et *LUFJ*, nous nous concentrons sur les problématiques liées à la gestion de la mémoire, et aborderons dans le chapitre suivant les spécificités liées au parallélisme.

Ce qui est important dans ce chapitre, ce ne sont pas directement les résultats, mais la méthodologie employée, et les différents prédicats qui seront réutilisés par la suite. Une approche plus simple aurait été de réserver une zone mémoire pour l'instrumentation, et rendre déterministe l'allocation. Le fait de ne pas introduire de telles contraintes rend les résultats plus réalistes mais plus difficiles à obtenir.

Dans ce cadre simplifié par rapport à la compilation de*AFJ* vers *LUFJ*, mais qui est proche des manipulations séquentielles des structures de bulles et de listes de processus de la compilation de AFJ vers LUFJ, nous définissons la notion d'équivalence

entre deux états du programme (décomposée en équivalence de blocs d'instructions, d'environnements locaux, et de tas). Nous débutons par une description de la syntaxe et de la sémantique. Nous continuons ensuite par la transformation, et enfin nous établissons une preuve de préservation sémantique pour cette transformation.

8.1 Syntaxe et sémantique du langage

L'ensemble des instructions regroupe les instructions classiques d'un langage impératif.

$$\begin{array}{rcl} c & ::= & x := e \mid x := y[e'] \mid x[e'] := e'' \\ & & \mid x := \texttt{allocate}(e) \mid \texttt{dispose}(e) \\ & & \mid m(e) \\ & & \mid \texttt{if } e \texttt{ then } b \texttt{ else } b \\ & & \mid \texttt{while } b \texttt{ do } b \end{array}$$

Ces instructions sont organisées en blocs, qui sont des listes d'instructions.

$$\begin{array}{rcl} b & ::= & [] \\ & & \mid c; b \end{array}$$

Par abus de notation nous notons $b_1; b_2$ la concaténation de deux listes d'instructions.

Une méthode est identifiée par son nom, un argument et un bloc d'instructions. Un programme est une suite de méthodes et un bloc d'instructions représentant le programme principal.

$$\begin{array}{rcl} \mathit{mth} & ::= & \underline{m(x)}\ b \\ \mathit{program} & ::= & \overline{\mathit{mth}}\ b \end{array}$$

Les valeurs de bases sont les entiers, les adresses logiques et les booléens:

$$\mathcal{V} \quad ::= \quad n \mid \ell \mid b \text{ où } n \in \mathbb{N}, \ell \in \mathbb{L}, \text{ et } b \in \mathbb{B}$$

Nous reprenons la représentation des adresses détaillées à la section 6.3. On distingue les adresses logiques ($\mathbb{L}$) des adresses physiques ($\mathbb{A}$). On distingue une adresse physique a_0, et on note $\mathbb{A}^* = \mathbb{A}\backslash\{a_0\}$ Une adresse logique est une paire composée d'une adresse physique et d'un entier représentant le nombre de fois où l'adresse physique a été allouée. Ainsi deux allocations sur la même adresse physique peuvent être distinguées. Cette distinction nous sera utile pour traiter les situations problématiques où un pointeur vers une adresse désallouée, redevient valide suite à une nouvelle allocation à cette même adresse.

$$\ell ::= (a, n) \text{ où } a \in \mathbb{A}, n \in \mathbb{N}$$

Les constantes sont composées des entiers et de l'adresse $(a_0, 0)$. Il est à noter que les adresses différentes de `null` n'en font pas parties, et ne sont donc pas manipulables par l'utilisateur, afin de faciliter les preuves sur la mémoire.

$$const \quad ::= \mathbb{N} \mid \mathbb{B} \mid \texttt{null}$$

Les expressions sont les constantes, les variables et les fonctions appliquées aux expressions. Les fonctions considérées sont les quatre opérations sur les entiers, l'égalité sur les entiers, les booléens et les adresses logiques, et l'opérateur de comparaison $<$ sur les entiers.

$$e \quad = \quad const \mid x \mid f\bar{e} \text{ où } x \in \mathcal{X}$$

L'environnement local est une fonction partielle des variables $\mathcal{X}$ vers les valeurs:

$$\rho : \mathcal{X} \quad \rightharpoonup \quad \mathcal{V}$$

Le tas est une fonction partielle prenant un emplacement mémoire et un décalage et retournant une valeur. La valeur d'une cellule mémoire non-allouée est indéfinie.

$$\sigma : \mathbb{L} \rightharpoonup \mathbb{N} \rightharpoonup \mathcal{V}$$

L'évaluation des expressions est définie de la façon suivante:

$$\begin{array}{rcl} [\![x]\!]_{\rho,\sigma} & = & \rho(x) \\ [\![v]\!]_{\rho,\sigma} & = & v \\ [\![fe_1 \ldots e_2]\!]_{\rho,\sigma} & = & [\![f]\!]_{\rho,\sigma}[\![e_1]\!]_{\rho,\sigma} \ldots [\![e_n]\!]_{\rho,\sigma} \end{array}$$

L'évaluation de l'égalité entre adresses est particulières. En effet on veut comparer uniquement des adresses qui correspondent à la dernière allocation sur l'adresse physique, autrement dit ces adresses doivent appartenir à l'ensemble des adresses logiques valides du tas. La comparaison impliquant une adresse non valide ne peut pas être évaluée.

$$\frac{\ell \in valide(\sigma)}{[\![\ell = \ell]\!] = true}$$

$$\frac{\ell \neq \ell' \ \wedge \ \ell, \ell' \in valide(\sigma)}{[\![\ell = \ell']\!] = false}$$

Nous pouvons maintenant définir l'état d'un programme, noté Σ :

$$\Sigma ::= \overline{mth}, L, \sigma, \mathcal{A}$$

où L est une pile de paires constituées d'un bloc d'instructions et d'un environnement local. L'élément en tête de pile, correspond au coupe bloc d'instructions, environnement local actif. La troisième composante, σ, est le tas mémoire. La quatrième $\mathcal{A}$, est une fonction qui stocke le nombre d'allocations pour chaque adresse physique.

Un état peut être indicé par S pour désigner un état du programme source, ou par T pour le programme compilé. Cet indice est également appliqué à la pile d'instruction, au tas et à $\mathcal{A}$.

La sémantique du langage est détaillée dans la figure 8.1. Il est à noter que seules la règles (**alloc**) fait apparaître la fonction $\mathcal{A}$ dans l'état. Elle est également présente pour toutes les autres règles, mais elle n'est pas utilisée. Nous avons choisie de ne pas la faire apparaître.

La règle d'allocation (**alloc**) choisit une nouvelle adresse physique a dans $\mathbb{A}^*$. Le compteur associé a cette adresse physique est incrémenté de un dans le nouveau tas. L'ancien compteur est à 0 si l'adresse physique a n'est pas présente dans le domaine du tas. Si l'adresse est présente, l'ancien numéro d'allocation est récupéré grâce à la fonction $\mathcal{A}()$.

La règle de lecture (**get**) assure avec la condition que ℓ est bien dans le domaine, et que le décalage n est bien inférieur au nombre de cellules allouées à cette adresse.

La règle d'écriture (**put**) s'assure également que l'adresse et le décalage sont correctement définis. La notation utilisée $\sigma \cdot [\ell, n \mapsto v]$ indique qu'à l'adresse ℓ et au décalage n la nouvelle valeur est v.

Les autres règles sont plus classiques.

8.2 Abstractions du tas

La comparaison du tas nécessite de créer une nouvelle représentation plus abstraite de celui-ci. En effet deux tas peuvent être identiques à un renommage des adresses près. C'est pourquoi, dans la nouvelle représentation, les adresses sont remplacées par un nouveau type de données. Cette représentation est indépendante du langage, elle est d'ailleurs utilisée dans le chapitre suivant.

La fonction qui calcule cette nouvelle représentation, *reachableGraph*, nécessite la définition de nouvelles fonctions.

Nous supposons un ensemble de noms de variable réservés (par exemple pour la compilation, cela sera les noms de variables utilisées uniquement dans le code instrumentée). Nous définissons à partir d'un environnement local ρ, la séquence de variables $\mathit{rootsVariables}(\rho)$ comme étant les variables dont le nom n'est pas réservé et qui contiennent une adresse.

Nous définissons également à partir d'un environnement local ρ et une séquence de variables $\overline{x}$ la séquence d'adresse $\mathit{rootsAddresses}_\rho(\overline{x})$ comme étant les adresses directement accessibles depuis les variables.

Le graphe mémoire atteignable ($\mathit{reachableGraph}_\sigma(L, F)$) est une représentation de la mémoire accessible depuis une liste d'adresses racines L, en dehors des adresses exclues contenues dans F. Il est représenté par une liste d'association constituée de paires, chemin et liste de valeurs accessible par ce chemin. Un chemin est une représentation d'une adresse, et est constitué d'une liste d'entier. Par exemple, le chemin 0.1 représente la cellule accessible depuis la deuxième case du contenu pointé par

$$\overline{mth},\ (x := e; b,\ \rho)\cdot L,\ \sigma \xrightarrow{\tau} \overline{mth},\ (b,\ \rho[x \mapsto v])\cdot L,\ \sigma \qquad \textbf{(assign)}$$
$$\text{si } [\![e]\!]_{\rho,\sigma} = v$$

$$\overline{mth},\ (x := \texttt{allocate}(e); b,\ \rho)\cdot L,\ \sigma, \mathcal{A} \xrightarrow{\texttt{alloc}\ \ell\ n} \overline{mth},\ (b,\ \rho[x \mapsto \ell]),\ \sigma\cdot[\ell \mapsto [0]^n],\ \mathcal{A}\cdot[a \mapsto \mathcal{A}(a)+1] \qquad \textbf{(alloc)}$$
$$\text{si } [\![e]\!]_{\rho,\sigma} = n,\ n > 0,\ \forall n, (a,n) \notin dom(\sigma),\ \ell = (a, \mathcal{A}(a)+1)$$

$$\overline{mth},\ (\texttt{dispose}(e); b,\ \rho)\cdot L,\ \sigma \xrightarrow{\texttt{free}\ \ell} \overline{mth},\ (b,\ \rho)\cdot L,\ \sigma_{|dom(\sigma)-\{\ell\}} \qquad \textbf{(dispose)}$$
$$\text{si } [\![e]\!]_{\rho,\sigma} = \ell \text{ et } \ell \in dom(\sigma)$$

$$\overline{mth},\ (x := y[e]; b,\ \rho)\cdot L,\ \sigma \xrightarrow{\texttt{read}\ \ell\ n\ v} \overline{mth},\ (b,\ \rho[x \mapsto v])\cdot L,\ \sigma \qquad \textbf{(get)}$$
$$\text{si } \rho(y) = \ell,\ [\![e]\!]_{\rho,\sigma} = n,\ \sigma(\ell)(n) = v$$

$$\overline{mth},\ (x[e_1] := e_2; b,\ \rho)\cdot L,\ \sigma \xrightarrow{\texttt{write}\ \ell\ n\ v} \overline{mth},\ (b,\ \rho)\cdot L,\ \sigma\cdot[\ell, n \mapsto v] \qquad \textbf{(put)}$$
$$\text{si } \rho(x) = \ell,\ [\![e_1]\!]_{\rho,\sigma} = n, [\![e_2]\!]_{\rho,\sigma} = v \text{ et } \ell \in dom(\sigma) \text{ et } \exists v_0, \sigma(\ell)(n) = v_0$$

$$\overline{mth},\ (\texttt{while}\ e\ \texttt{do}\ b; b',\ \rho)\cdot L,\ \sigma \xrightarrow{\tau} \overline{mth},\ (b; \texttt{while}\ e\ \texttt{do}\ b; b',\ \rho)\cdot L,\ \sigma \qquad \textbf{(loop}_\textbf{t}\textbf{)}$$
$$\text{si } [\![e]\!]_{\rho,\sigma} = \texttt{true}$$

$$\overline{mth},\ (\texttt{while}\ e\ \texttt{do}\ b; b',\ \rho)\cdot L,\ \sigma \xrightarrow{\tau} \overline{mth},\ (b',\ \rho)\cdot L,\ \sigma \qquad \textbf{(loop}_\textbf{f}\textbf{)}$$
$$\text{si } [\![e]\!]_{\rho,\sigma} = \texttt{false}$$

$$\overline{mth},\ (\texttt{if}\ e\ \texttt{then}\ b_1\ \texttt{else}\ b_2; b,\ \rho)\cdot L,\ \sigma \xrightarrow{\tau} \overline{mth},\ (b_1; b,\ \rho)\cdot L,\ \sigma \qquad \textbf{(cond}_\textbf{t}\textbf{)}$$
$$\text{si } [\![e]\!]_{\rho,\sigma} = \texttt{true}$$

$$\overline{mth},\ (\texttt{if}\ e\ \texttt{then}\ b_1\ \texttt{else}\ b_2; b,\ \rho)\cdot L,\ \sigma \xrightarrow{\tau} \overline{mth},\ (b_2; b,\ \rho)\cdot L,\ \sigma \qquad \textbf{(cond}_\textbf{f}\textbf{)}$$
$$\text{si } [\![e]\!]_{\rho,\sigma} = \texttt{false}$$

$$\overline{mth},\ (m(e); b',\ \rho)\cdot L,\ \sigma \xrightarrow{\tau} \overline{mth},\ (b,\ [x \mapsto v])\cdot(b',\ \rho)\cdot L,\ \sigma \qquad \textbf{(methodcall)}$$
$$\text{si } [\![e]\!]_{\rho,\sigma} = v, m(x)\ b \in \overline{mth}$$

$$\overline{mth},\ ([],\ \rho)\cdot(b',\ \rho')\cdot L,\ \sigma \xrightarrow{\tau} \overline{mth},\ (b',\ \rho')\cdot L,\ \sigma \qquad \textbf{(endmethod)}$$

FIGURE 8.1 – *Sémantique opérationnelle*

la première adresse de la liste d'adresses racines. Il est à noter que nous n'adoptons pas la notation habituelle des listes pour représenter un chemin, nous utilisons un . pour séparer les éléments. Le 0 représente le chemin de la première adresse de la liste d'adresses racines. Le 1 est utilisé car l'adresse est présente dans la deuxième case.

Les adresses présentes dans le tas et dans la liste des adresses exclues sont remplacées par $\perp$. Dans le tas, il peut exister des liens qui n'en sont pas vraiment si l'on ne prend en compte que les adresses physique (i.e. sans les numéros d'allocation). Une adresse peut être présente dans une cellule du tas, mais n'être plus allouée. Si ensuite une nouvelle allocation réalloue cette adresse, un lien entre cette cellule et cette nouvelle adresse est créée si l'on ne prend compte que les adresse physiques. Nous ne voulons pas représenter un tel lien, qui n'a pas réellement de sens. Une autre allocation, aurait pu choisir une adresse physique différente, et le lien n'aurait donc pas existé. C'est pourquoi les adresses présentes dans le tas dont le numéro d'allocation est strictement inférieur à celui de l'adresse dans le domaine sont également remplacées par $\perp$.

Un exemple d'une telle transformation est présentée dans la figure 8.2. Dans cet exemple, nous disposons d'un environnement local ρ, et d'un tas σ. Le but est de calculer le graphe atteignable à partir des adresses contenues dans les variables x et y, et en excluant l'adresse contenue dans la variable a. Le calcul débute par l'analyse de la mémoire accessible à partir de l'adresse $\rho(x)$, soit ℓ_1. Comme nous ne voulons pas les adresses réelles dans notre tas symbolique, pour ne pas dépendre de l'allocation, nous les remplaçons par des adresses symboliques. Comme ℓ_1 est la première adresse que nous considérons, elle est remplacée par *0*. Les valeurs entières accessibles ne sont pas modifiées. La valeur ℓ_3 est une adresse, elle doit donc être remplacée par son adresse symbolique. Comme nous n'avons jamais rencontré cette adresse, nous devons calculer son adresse symbolique. Cette adresse symbolique est formée par l'adresse symbolique de la zone mémoire actuellement considérée, *0*, concaténée à la position de la case mémoire, soit 3 : l'adresse symbolique de ℓ_3 est donc *0.3*. L'adresse ℓ_3 est une adresse accessible, il faudra donc explorer le contenu du tas accessible à partir de cette adresse. L'analyse du tas se poursuit avec l'adresse contenue dans la variable y, soit ℓ_2. En explorant le tas accessible à cette adresse, nous retrouvons l'adresse ℓ_1, que nous avons déjà rencontrée. Nous la remplaçons donc par son adresse symbolique *0*. Nous rencontrons ensuite l'adresse ℓ_4. Cependant, cette adresse fait partie des adresses exclues données en paramètres à la fonction. Nous ne devons donc pas la suivre, nous la remplaçons donc par $\perp$. L'analyse se termine avec l'exploration du contenu accessible à partir de l'adresse ℓ_3, car aucune nouvelle adresse n'est découverte.

L'intérêt de cette représentation est que pour une autre exécution du programme, les adresses allouées peuvent être différentes, mais le graphe sera identique. Par exemple, pour l'environnement mémoire présentée dans la figure 8.3 qui diffère uniquement par les valeurs des adresses de l'exemple 8.2, le graphe mémoire atteignable est identique.

Par abus de notation nous utilisons cette définition en remplaçant la séquence

$$\begin{array}{lcl}\rho & = & \{x \mapsto \ell_1, y \mapsto \ell_2, a \mapsto \ell_4\} \\ \sigma & = & \ell_1 \mapsto [0;1;2;\ell_3], \ell_2 \mapsto [0;0;\ell_1;\ell_4], \ell_3 \mapsto [0;2;\ell_2;\ell_1], \ell_4 \mapsto [0;1;2;\ell_3]\end{array}$$

$$reachableGraph_\sigma(rootsAddresses_\rho([x;y]), \{\rho(a)\}) = \begin{array}{rcl} 0 & \mapsto & [0;1;2;0.3], \\ 1 & \mapsto & [0;0;0;\bot], \\ 0.3 & \mapsto & [0;2;1;0] \end{array}$$

FIGURE 8.2 – *Exemple de représentation d'un graphe atteignable*

$$\begin{array}{lcl}\rho & = & \{x \mapsto \ell'_1, y \mapsto \ell'_2, a \mapsto \ell'_4\} \\ \sigma & = & \ell'_1 \mapsto [0;1;2;\ell'_3], \ell'_2 \mapsto [0;0;\ell'_1;\ell'_4], \ell_3 \mapsto [0;2;\ell'_2;\ell'_1], \ell'_4 \mapsto [0;1;2;\ell'_3]\end{array}$$

FIGURE 8.3 – *Environnement mémoire*

d'adresses en paramètre par un environnement local. Elle est définie ainsi:

$$reachableGraph_\sigma(\rho, F) = reachableGraph_\sigma(rootsAddresses_\rho(rootsVariables(\rho)), F)$$

$$\begin{array}{rcl} \mathbb{P} & ::= & list\ \mathbb{N} \\ RV & ::= & \mathbb{Z} \cup \mathbb{B} \cup \mathbb{P} \cup \{\bot\} \\ RGA & ::= & \mathbb{P} \rightharpoonup \overline{RV} \\ SHA & ::= & \mathbb{L} \rightharpoonup \mathbb{P} \end{array}$$

Nous définissons également le tas symbolique $symAddr(\rho, \sigma, F)$ comme étant la liste d'association entre les adresses du tas et leur représentation sous la forme d'un chemin. Ce tas est calculé lors du calcul du graphe mémoire.

Sur l'exemple 8.2, le tas symbolique est le suivant

$$symAddr(\sigma, \rho, \rho(a)) = \ell_1 \mapsto 0, \ell_2 \mapsto 1, \ell_3 \mapsto 0.3$$

Enfin nous définissons la fonction du graphe mémoire généralisé qui calcule à la fois le graphe mémoire et le tas symbolique.

$$reachableGraphGen_\sigma(\rho, F) = (reachableGraph_\sigma(\rho, F), symAddr(\sigma, \rho, F))$$

En terme de calcul, la fonction *reachableGraph* et *symHeap* sont les projections du résultat de la fonction *reachableGraphGen*.

A partir de ce tas symbolique, nous définissons la fonction $G^L_\sigma(\ell)$ qui à partir d'un tas σ, d'une liste d'adresses exclues L, retourne le chemin correspondant à l'adresse donnée ℓ.

$$
\begin{array}{rcl}
reachableGraphGen_{\sigma}(\rho, F) & = & reachGraph(\sigma, rootsAddresses_{\rho}(rootsVariables(\rho)), F, [][]) \\
\\
reachGraph & & (h : \sigma, L : \overline{\mathbb{L}}, F : \overline{\mathbb{L}}, g : RGA, s : SHA) : (RGA, SHA) \\
& = & match\ L \backslash F\ with \\
& & \mid a :: l' \Rightarrow let(ch_a, s_1) = \\
& & \qquad if(a\ \in\ dom(s))\ then\ (s(a), s) \\
& & \qquad else\ let\ nc = (|dom(s)| + 1)\ in\ (nc, s \cdot [a \mapsto nc])\ in \\
& & \qquad\quad let(tab, s', l_1, \cdot) = tabGraph(a, h, F, s_1)\ in \\
& & \qquad\quad let\ g' = add(g, ch_a, tab)\ in \\
& & \qquad\quad reachGraph(h, l' +\!\!+ l_1, F, g', s') \\
& & \mid [] \Rightarrow (g, s) \\
\\
tabGraph & & (a : \mathbb{L}, h : \sigma, F : \overline{\mathbb{L}}, s : SHA) : \overline{RV} \times SHA \times \overline{\mathbb{L}} \times \mathbb{N} \\
& = & fold\_left\ (f\ F\ s(a))\ ([], s, [], 0)\ \sigma(a) \\
\\
f & & (F : \overline{\mathbb{L}}, ch : \mathbb{P}, (tab, s, l, n) : \overline{RV} \times SHA \times \overline{\mathbb{L}} \times \mathbb{N}, v : \mathcal{V}) : \\
& & \overline{RV} \times SHA \times \overline{\mathbb{L}} \times \mathbb{N} \\
& = & if\ isAddress(v)\ then \\
& & \quad if(v\ \in\ F)\ then\ (tab :: \bot, s, l, S(n)) \\
& & \quad else\ if(v\ \in\ dom(s))\ then\ (tab :: s(v), s, l, S(n)) \\
& & \qquad else\ let\ ch_v = ch \cdot n\ in \\
& & \qquad\quad (tab \cdot ch_v, s \cdot [v \mapsto ch_v]), l \cdot v, S(n)) \\
& & else(tab \cdot v, s, l, S(n))
\end{array}
$$

Définition 8.1 *Une adresse l appartient à un tas σ, noté $l \in \sigma$, si $l \in dom(\sigma)$ ou $\exists x, i$ tel que $\sigma(x)[i] = l$*

Définition 8.2 *Soient deux environnements locaux ρ_1, ρ_2 et deux tas σ_1, σ_2, et deux ensembles d'adresses F_1, F_2 tels que $reachableGraph_{\rho_1}(\sigma_1, F_1) = reachableGraph_{\rho_2}(\sigma_2, F_2)$, et deux adresses l_1, l_2 telles que $l_1 \in \sigma_1 \wedge l_2 \in \sigma_2$, et $l_1 \notin F_1 \wedge l_2 \notin F_2$. Ces deux adresses sont équivalentes, noté $l_{1\rho_1,\sigma_1,F_1} \sim^{add}_{\rho_2,\sigma_2,F_2} l_2$, si $symAddr(\rho_1, \sigma_1, \emptyset)(l_1) = symAddr(\rho_2, \sigma_2, \emptyset)(l_2)$*

Définition 8.3 *Étant donné deux environnements locaux ρ_1, ρ_2 et deux tas σ_1, σ_2 et deux ensembles d'adresses F_1, F_2, tels que $reachableGraph_{\rho_1}(\sigma_1, F_1) = reachableGraph_{\rho_2}(\sigma_2, F_2)$, deux valeurs v_1, v_2 sont équivalentes, notés $v_{1\rho_1,\sigma_1,F_1} \equiv^{val}_{\rho_2,\sigma_2,F_2} v_2$ si :*

- *$v_1 = v_2$ si v_1 et v_2 ne sont pas des adresses.*
- *$v_{1\rho_1,\sigma_1,F_1} \sim^{add}_{\rho_2,\sigma_2,F_2} v_2$ si v_1 et v_2 sont des adresses.*

Définition 8.4 *Étant donné deux environnements locaux ρ_1, ρ_2 et deux tas σ_1, σ_2 et deux ensembles d'adresses F_1, F_2 tels que $reachableGraph_{\rho_1}(\sigma_1, \emptyset) = reachableGraph_{\rho_2}(\sigma_2, \emptyset)$, deux expressions e_1, e_2 sont équivalentes, notés $e_{1\rho_1,\sigma_1,F_1} \sim^{exp}_{\rho_2,\sigma_2,F_2} e_2$ si $\exists v_1, [\![e_1]\!]_{\rho_1,\sigma_1} = v_1$, $\exists v_2, [\![e_2]\!]_{\rho_2,\sigma_2} = v_2$ et $v_{1\rho_1,\sigma_1,F_1} \equiv^{val}_{\rho_2,\sigma_2,F_2} v_2$*

Lemmes sur le graphe mémoire :

Le tas évolue suite à certaines règles de sémantique, et donc le graphe mémoire évolue également. Nous nous intéressons à l'ajout et la suppression d'élément dans le graphe et l'impact sur l'équivalence.

Étant donné deux graphes atteignables égaux, l'ajout d'adresses pointant vers des valeurs égales, à la même position dans la séquence d'adresses, conserve la relation d'équivalence.

Lemme 8.1 *Soit deux séquences d'adresses L_1, L_2 de même taille, deux tas σ_1, σ_2 et deux ensembles d'adresses F_1, F_2, tel que $reachableGraph_{\sigma_1}(L_1, F_1) = reachableGraph_{\sigma_2}(L_2, F_2)$, et deux adresses l_1, l_2 telles que $l_1 \notin dom(\sigma_1)$, $l_2 \notin dom(\sigma_2)$.*

Soit $\sigma_1' = \sigma_1 \cdot [l_1 \mapsto [v_1; \ldots; v_n]]$ et $\sigma_2' = \sigma_2 \cdot [l_2 \mapsto [v_1; \ldots; v_n]]$.

Soit L_1', L_2' les deux séquences d'adresses issues de l'insertion à la même position de l_1, l_2 dans respectivement L_1 et L_2.

Nous avons alors $reachableGraph_{\sigma_1'}(L_1', F_1) = reachableGraph_{\sigma_2'}(L_2', F_2)$.

Esquisse de preuve. On sait que le tas est modifié par l'ajout de cellule. Ces cellules sont identiques dans les deux cas. Quelles sont les modifications apportées par cet ajout. Il n'y a pas de nouveau lien partant de ces nouvelles cellules, car, elles ne contiennent pas d'adresses. Peut-il y avoir des liens vers la nouvelle adresse ? Des cellules dans le tas peuvent contenir l'adresse qui a été allouée. Ces adresses ont nécessairement été stockées avant l'allocation, et ont donc un numéro d'allocation inférieur à la nouvelle adresse. Il n'y a donc pas de lien vers cette nouvelle adresse. L'équivalence est conservée. □

Étant donné deux graphes atteignables équivalents, la suppression d'adresse équivalente dans les deux tas conserve-t-elle l'équivalence ?

Lemme 8.2 *Soit deux environnements locaux ρ_1, ρ_2, deux adresses ℓ_1, ℓ_2, deux tas σ_1, σ_2, et deux ensembles d'adresses F_1, F_2 et deux graphes mémoire G_1, G_2 tel que $G_1 = reachableGraph_{\sigma_1}(\rho_1, F_1)$ et $G_2 = reachableGraph_{\sigma_2}(\rho_2, F_2)$, $G_1 = G_2$ et $l_{1\rho_1,\sigma_1,F_1} \sim^{add}_{\rho_2,\sigma_2,F_2} l_2$.*

Nous avons alors $reachableGraph_{\sigma_1 \cdot [\ell_1 \mapsto \perp]}(\rho_1, F_1) = reachableGraph_{\sigma_2 \cdot [\ell_2 \mapsto \perp]}(\rho_2, F_2)$

Pour toutes adresses logiques valides, qui n'appartiennent pas à la liste d'exclusion, on a équivalence entre ces adresses et les chemins du graphe atteignable.

Lemme 8.3 *Étant donné une liste d'adresses F, et un tas σ, alors $\forall \ell_1, \ell_2$ telles que $\ell_1 \in valide(\sigma) \wedge \ell_2 \in valide(\sigma) \wedge \ell_1 \notin F \wedge \ell_2 \notin F \wedge \ell_1 = \ell_2 \Leftrightarrow G^F_\sigma(\ell_1) = G^F_\sigma(\ell_2) \wedge G^F_\sigma(\ell_1) \neq \perp \wedge G^F_\sigma(\ell_2) \neq \perp$*

Pour tout chemin dans le graphe atteignable, il existe une adresse logique valide n'appartenant pas à la liste d'exclusion correspondant à ce chemin.

Lemme 8.4 *Étant donné une liste d'adresses F, et un tas σ, alors $\forall \ell^c \in G^F_\sigma()$, $\exists \ell^l$, $\ell^l \in valide(\sigma) \wedge \ell^l \notin F, \wedge G^F_\sigma(\ell^c) = \ell^l$*

Une adresse non valide ou appartenant à la liste d'exclusion est équivalent au fait que dans le graphe atteignable cette adresse soit associée à $\bot$.

Lemme 8.5 *Étant donné une liste d'adresses L, et un tas σ, alors $\forall \ell \bot \in valide(\sigma) \vee \ell \in L \Leftrightarrow G^L_\sigma(\ell) = \bot$.*

Esquisse de preuve.

— $\Rightarrow$: On sait par définition du graphe atteignable que les adresses non valides ou appartenant à la liste d'exclusion sont associées à $\bot$

— $\Leftarrow$: par définition du graphe atteignable

□

8.3 Transformation source à source

Les langages source et cible sont identiques. La compilation a pour but d'ajouter les instructions nécessaires afin d'avoir en mémoire une liste chaînée de pointeurs vers les zones mémoires allouées par le programme. Les instructions qui nécessitent une compilation sont donc l'allocation et la libération mémoire afin de mettre à jour la liste. L'appel de méthode, et le début de chaque méthode sont également modifiés pour transmettre dans l'environnement mémoire de la nouvelle méthode, la référence à la liste chaînée représentant la mémoire. Il y a également une phase d'initialisation de la structure chaînée en début de programme.

$$
\begin{aligned}
\mathcal{C}(\overline{\text{mth}}\ b) &= \overline{\mathcal{C}(\text{mth})} initProgram; \mathcal{C}(b) \\
\mathcal{C}(c; b) &= \mathcal{C}(c); \mathcal{C}(b) \\
\mathcal{C}([]) &= [] \\
\mathcal{C}(x := \texttt{allocate}(e)) &= x := \texttt{allocate}(e); addList(x, e) \\
\mathcal{C}(\texttt{dispose}(e)) &= \texttt{dispose}(e); removeList(e) \\
\mathcal{C}(m(x)b) &= \underline{m}(arg) initMethod(x); \mathcal{C}(b) \\
\mathcal{C}(m(e)) &= prepareCallMethod(e); \underline{m}(arg) \\
\mathcal{C}(c) &= c \text{ sinon}
\end{aligned}
\tag{8.1}
$$

Les suites d'instructions *initProgram*, *prepareCallMethod*, *initMethod*, *addList*, *removeList* sont des blocs d'instructions supplémentaires permettant de gérer la liste chaînée représentant la mémoire. Certains de ses blocs sont paramétrés par une variable ou une expressions (ou les deux). Ces instructions sont visibles dans l'annexe A.1. Nous désignons l'un de ces blocs d'instructions par le terme *code instrumenté*.

$$
\begin{aligned}
\textit{code instrumenté} \quad := \quad & \{initProgram, prepareCallMethod(e), initMethod(x), \\
& addList(x, e), removeList(e)\}
\end{aligned}
$$

Les noms des variables réservés sont les suivantes :

$$(arg, mExpr, size, current, oldCurrent, addrRm, node, tmp)$$

Les programmes sources sont supposés ne pas utiliser ces variables réservées.

8.4 Correction de la transformation

8.4.1 Équivalence d'états

Nous présentons dans un premier temps les différentes définitions liées à l'équivalence entre états. Cette équivalence est la base du raisonnement de correction de la compilation.

Nous définissons une relation d'équivalence, entre un état cible Σ_T et un état source Σ_S et la notons $\Sigma_T \sim \Sigma_S$.

L'équivalence d'états se décompose en trois sous équivalences : instructions, environnement local et tas. Pour comparer les tas, il est nécessaire d'avoir l'environnement mémoire, la comparaison de tas se décompose donc en fonction du nombre d'environnements locaux.

Nous notons ρ^i (respectivement b^i) l'environnement local (respectivement le bloc d'instructions) en position i dans la pile. Nous avons une pile de couple, mais nous pouvons également le voir comme un couple de pile.

Définition 8.5 *Compatibilité entre $\mathcal{A}$ et le tas.*

$$\frac{dom(\mathcal{A}) = dom(\sigma) \wedge \forall (a,n) \in dom(\sigma), n = \mathcal{A}(a)}{\mathcal{A} \text{ compatible } \sigma}$$

Définition 8.6 *Équivalence d'état mémoire*

$$\frac{\begin{array}{c}|L_T| = |L_S| \quad \wedge \quad L_T, \sigma_T \sim^{instr} L_S, \sigma_S \wedge \quad L_T \sim^{env} L_S \\ \wedge \forall i, i < |L_T| \quad \rho_T^i, \sigma_T \sim^{heap} \rho_S^i, \sigma_S \wedge \mathcal{A}_T \text{ compatible } \sigma_T \wedge \mathcal{A}_S \text{ compatible } \sigma_S\end{array}}{L_T, \sigma_T, \mathcal{A}_T \sim L_S, \sigma_S, \mathcal{A}_S}$$

Équivalence instructions Afin d'identifier les instructions instrumentées des instructions présentes dans le code source nous les colorons, ■ pour les premières, et ■ pour les secondes. Pour identifier une instruction instrumentée, nous inspectons la présence ou non d'une variable réservée dans l'instruction. Si c'est le cas, l'instruction est instrumentée.

L'équivalence de piles d'instructions s'effectue en comparant chaque niveau de la pile. Les deux piles doivent donc être de taille identique. Pour être comparé le bloc d'instructions instrumentées est transformé par la fonction $d()$ (définition 8.10). Ces instructions comportent des expressions, et afin de les comparer il est nécessaire de les évaluer et donc d'avoir les environnements locaux et les tas source et cible en paramètre. Cette comparaison d'instruction $\sim$ est définie dans la définition 8.8.

Définition 8.7 *Équivalence pile d'instruction*

$$\frac{|L_T| = |L_S| \wedge \forall i, i < |L_T| \quad d(b_T^i) \sim_{\rho_T^i \rho_S^i}^{\sigma_T \sigma_S} b_S^i}{L_T, \sigma_T \ \sim^{instr} \ L_S, \sigma_S}$$

L'équivalence de bloc se base sur l'équivalence d'instruction. L'équivalence d'instruction est liée à la compilation et à la manière dont sont transformées les instructions avec la fonction $d()$.

La relation d'équivalence sur les instructions se base sur un filtrage des instructions, l'ordre compte.

Définition 8.8 *Équivalence d'instruction :*

$$\begin{array}{rll}
m(e) & \sim_{\rho_T \rho_S}^{\sigma_T \sigma_S} & m(e) \text{ où } e \neq mExpr \\
m(mExpr) & \sim_{\rho_T \rho_S}^{\sigma_T \sigma_S} & m(e) \text{ où } [\![e]\!]_{\rho_S, \sigma_S} = \rho_T(mExpr) \\
\texttt{if } e \texttt{ then } b_1 \texttt{ else } b_2 & \sim_{\rho_T \rho_S}^{\sigma_T \sigma_S} & \texttt{if } e \texttt{ then } b_1' \texttt{ else } b_2' \textit{ si } b_1 \sim_{\rho_T \rho_S}^{\sigma_T \sigma_S} b_1' \\
 & & \wedge\ b_2 \sim_{\rho_T \rho_S}^{\sigma_T \sigma_S} b_2' \\
\texttt{while } e \texttt{ do } b & \sim_{\rho_T \rho_S}^{\sigma_T \sigma_S} & \texttt{while } e \texttt{ do } b' b \sim_{\rho_T \rho_S}^{\sigma_T \sigma_S} b' \\
c & \sim_{\rho_T \rho_S}^{\sigma_T \sigma_S} & c' \textit{ si } c = c' \textit{pour tous les autres cas}
\end{array}$$

La relation d'équivalence de blocs d'instructions est la suivante :

$$\begin{array}{rll}
[] & \sim_{\rho_T \rho_S}^{\sigma_T \sigma_S} & [] \\
c_1; b_1 & \sim_{\rho_T \rho_S}^{\sigma_T \sigma_S} & c_2; b_2 \textit{ si } c_1 \sim_{\rho_T \rho_S}^{\sigma_T \sigma_S} c_2 \textit{ et } b_1 \sim_{\rho_T \rho_S}^{\sigma_T \sigma_S} b_2
\end{array}$$

Définition 8.9 *La relation $suffixStrict(b_1, b_2)$ indique que le bloc d'instructions b_1 est un suffixe strict du bloc b_2.*

Définition 8.10 *La fonction $d()$ supprime les instructions instrumentées (de couleur ■) issues de la compilation d'un programme. Le cas des appels de méthode est particulier, car il nécessite le remplacement de l'expression en paramètre. Cette fonction est également un filtrage, l'ordre compte.*

$$\begin{array}{rcl}
d(prepareCallMethod(e); \underline{m}(arg)) & = & m(e) \\
d(b; \underline{m}(arg)) & = & m(mExpr) \text{ où } \exists e,\ suffixStrict(b, prepareCallMethod(e)) \\
d([]) & = & [] \\
d(\texttt{while } e \texttt{ do } b | \blacksquare; b_s) & = & (\texttt{while } e \texttt{ do } d(b)); d(b_s) \\
d(\texttt{if } e \texttt{ then } b_1 \texttt{ else } b_2 | \blacksquare; b_s) & = & (\texttt{if } e \texttt{ then } d(b_1) \texttt{ else } d(b_2)); d(b_s) \\
d(c | \blacksquare; b_s) & = & d(b_s) \\
d(c | \blacksquare; b_s) & = & c; d(b_s)
\end{array}$$

Équivalence environnements locaux L'équivalence de pile pour les environnements locaux s'effectue niveau par niveau, les deux piles doivent donc être de même taille. Le principe dans le cas général est de comparer uniquement les variables contenant des valeurs non adresse et dont les noms ne sont pas réservés. Cela est effectué par

la fonction *filterVar*(ρ) qui retourne l'environnement ρ restreint aux variables dont les noms ne sont pas réservés et contenant une valeur non adresse.

Dans le cas où l'on débute un appel de méthode, les environnements locaux du programme source et cible sont particuliers, l'argument de la méthode n'est pas localisé dans le même emplacement. La comparaison prend en compte ce cas particulier, et s'effectue sur le contenu des arguments des méthodes. Dans le programme source, pour une méthode $m(x)$ l'argument se trouve dans la variable x. Dans le programme cible, la méthode doit récupérer une information supplémentaire, l'adresse de la structure de données contenant les adresses allouées. Cet valeur doit être transmise en plus de l'argument original. Ces deux valeurs sont donc placées dans un tableau *arg*. L'argument de la méthode est donc l'adresse de ce tableau *arg*, qui contient dans la première case, l'adresse de la structure de données et dans la seconde l'argument original. Dans l'équivalence, nous vérifions donc que
$heapS(\rho_T(argM))(1) = \rho_S(x)$.

Définition 8.11 *Équivalence de piles d'environnement :*

$$\frac{\begin{array}{c}|L_T| = |L_S| \quad \forall i, i < |L_T| \\ (\forall x,\ instrTop_{L_T} = fst(initMethod(x)) \Rightarrow \exists v_S, v_T, \ell,\ \rho^i_S = [x \mapsto v_S] \wedge \\ \rho^i_T = [arg \mapsto \ell] \wedge \sigma_T(\rho^i_T(arg)[1]) = v_T \wedge v_{T\,\rho^i_T,\sigma_T,} \equiv^{val}_{\rho^i_S,\sigma_S,\emptyset} v_S) \wedge \\ (\forall x,\ instrTop_{L_T} \neq fst(initMethod(x)) \Rightarrow filterVar(\rho^i_S) = filterVar(\rho^i_T))\end{array}}{L_T \ \sim^{env} \ L_S}$$

Juste avant de définir l'équivalence de tas, nous introduisons quelques définitions auxiliaires.

Liste Nous souhaitons dans un état cible, avoir la liste des adresses allouées par le programme. Ces adresses sont stockées dans une liste. Nous définissons le prédicat *Liste*(σ, ℓ, xs) qui indique qu'à l'adresse ℓ dans le tas σ est accessible un triplet contenant une adresse, un entier et une adresse à partir de laquelle est accessible une liste.

Définition 8.12 *Le prédicat suivant Liste*(σ, ℓ, xs) *permet d'indiquer qu'à partir de l'adresse* ℓ *dans le tas* σ *est accessible une liste* xs*. Chaque maillon de cette liste est composé de trois éléments, une adresse, un entier et l'adresse pointant sur le maillon suivant.*

$$\frac{}{Liste(\sigma, \texttt{null}, [])}$$

$$\frac{Liste(\sigma, \ell_{xs}, xs) \quad \wedge \quad \sigma(\ell)[0] = (\ell_a) \quad \wedge \quad \sigma(\ell)[1] = n_a \quad \wedge \quad \sigma(\ell)[2] = (\ell_{xs})}{Liste(\sigma, \ell, (\ell_a, n_a, \ell_{xs}) :: xs)}$$

Nous définissons des fonctions auxiliaires permettant d'extraire d'une liste, certains éléments de chaque maillon (le pointeur vers le maillon suivant ou l'adresse stockée)

$$nodeList_\sigma(\ell) = \forall xs, Liste(\sigma, \ell, xs) \Rightarrow \ell :: (map\ trd\ xs)$$

$$infoList_\sigma(\ell) = \forall xs, Liste(\sigma, \ell, xs) \Rightarrow (map\ fst\ xs)$$

Il est à noter que trd et fst sont des fonctions appliquées à des triplets qui retournent respectivement le troisième et le premier élément.

Nous prouvons qu'une liste ne contient pas de doublons.

Lemme 8.6 *Étant donné un tas σ, une adresse ℓ et une liste xs, si l'on a $Liste(\sigma, \ell, xs)$ alors on a $NoDup(nodeList_\sigma(\ell))$*

Démonstration. La preuve se fait par contradiction, en supposant que la présence en double de la même adresse ℓ_a dans la liste $nodeList_\sigma(\ell)$. Cette adresse pointe nécessairement vers une liste par définition de *liste*.

On sait également que la taille de xs est finie. Donc pour toutes adresses ℓ_1, ℓ_2 telles que $\ell_1, \ell_2 \in nodeList_\sigma(\ell)$, il existe donc deux listes xs' et xs'' telles que $Liste(\sigma, \ell_1, xs')$ et $Liste(\sigma, \ell_2, xs'')$ avec $|xs'| > |xs''|$.

Or il n'existe qu'une seule liste à partir de ℓ_a, donc $|xs'| = |xs''|$, cela est en contradiction avec la remarque précédente.

□

Afin de désigner une instruction dans un bloc, nous définissons les positions. Une position est une liste d'entiers.

Nous disposons également d'une fonction partielle retournant l'instruction à une position donnée dans un bloc.

$$instrAt(b, pos)$$

$$\begin{aligned}
instrAt(c; b, 0) &= c \\
instrAt(c; b, (S\ n).pos') &= instrAt(b, n.pos') \\
instrAt(\texttt{if}\ \cdot\ \texttt{then}\ b_1\ \texttt{else}\ b_2; b, 0.0.pos) &= instrAt(b_1, pos) \\
instrAt(\texttt{if}\ \cdot\ \texttt{then}\ b_1\ \texttt{else}\ b_2; b, 0.1.pos) &= instrAt(b_2, pos) \\
instrAt(\texttt{while}\ \cdot\ \texttt{do}\ b; b', 0.pos) &= instrAt(b, pos)
\end{aligned}$$

Étant donné un état Σ, nous notons $instrTop_\Sigma$, l'instruction en première position du bloc au sommet de la pile des instructions.

Nous avons le prédicat qui permet de savoir si l'instruction active de l'état Σ est une instruction d'un bloc d'instructions instrumentées b, c'est à dire de l'instruction en position pos du bloc b.

$$instr_b(\Sigma, pos) \equiv instrAt(b, pos) = instrTop_\Sigma$$

La liste des adresses exclues est celle constituée des adresses allouées dans les instructions instrumentées, et qui permettent principalement de d'avoir la liste des

adresses allouées dans les instructions non instrumentées. Cette liste est accessible depuis $arg[0]$, elle est donc constituée de l'adresse de arg et de tous les pointeurs de la liste. Nous la notons $listp_{\sigma,\rho}$.

$$listp_{\sigma,\rho} = nodeList_{\sigma}(\sigma(\rho(arg))(0)) \cup \rho(arg)$$

Nous avons désormais toutes les définitions pour exprimer l'équivalence de tas.

Nous pouvons maintenant définir l'équivalence de tas comme l'égalité de graphe mémoire. On peut voir dans cette définition une conjonction de quatre propriétés. Grâce aux gardes, seule une propriété est valide à la fois. Le cas général est représenté par la dernière propriété. Le première et la seconde sont quasiment identiques au cas général, mais sont actives dans les blocs instrumentés *addList* et *removeList* respectivement. Dans ces blocs, une variable réservée n'est pas présente dans la liste d'exclusion classique, il est nécessaire de prendre en compte cet emplacement séparément. La troisième propriété représente le cas particulier de la première instruction *initMethod*, où l'argument de la méthode est situé dans deux emplacements différents.

Définition 8.13 *Équivalence du tas*

$$\frac{\begin{array}{c}(\exists pos, x, e\ instrTop_{\rho_T} = instrAt(addList(x,e), pos) \wedge pos > 0 \Rightarrow \\ reachableGraph_{\sigma_T}(\rho_T, listp_{\sigma,\rho} \cup \rho_T(node)) = reachableGraph_{\sigma_S}(\rho_S, \emptyset)) \wedge \\ \\ (\exists pos, x, e\ instrTop_{\rho_T} = instrAt(removeList(e), pos) \wedge pos > 0 \Rightarrow \\ reachableGraph_{\sigma_T}(\rho_T, listp_{\sigma,\rho} \cup \rho_T(current)) = reachableGraph_{\sigma_S}(\rho_S, \emptyset)) \wedge \\ \\ (\exists pos, x\ instrTop_{\rho_T} = instrAt(initMethod(x), pos) \Rightarrow \\ reachableGraph_{\sigma_T}(\sigma_T(\rho_T(arg))(1), listp_{\sigma,\rho}) = reachableGraph_{\sigma_S}(\rho_S, \emptyset)) \wedge \\ \\ (\forall pos_0, pos_1, x, e\ pos_1 > 0 \wedge instrTop_{\rho_T} \neq instrAt(addList(x,e), pos_1) \wedge \\ instrTop_{\rho_T} \neq instrAt(removeList(e), pos_1) \wedge \\ instrTop_{\rho_T} \neq instrAt(initMethod(x), pos_0) \Rightarrow \\ reachableGraph_{\sigma_T}(\rho_T, listp_{\sigma,\rho}) = reachableGraph_{\sigma_S}(\rho_S, \emptyset))\end{array}}{\rho_T, \sigma_T \ \sim^{heap}\ \rho_S, \sigma_S}$$

Définition 8.14 *Une instruction est présente dans une autre, noté $c_1 \in c_2$, si c_1 est un sous terme de c_2.*

On étend la notion de présence d'une instruction aux programmes, méthodes et aux états mémoires.

$$\begin{array}{rcl} c & \in & p\ si\ p = \bar{mth}\ c_{main}\ et\ c \in \bar{mth}\ ou\ c \in c_{main} \\ c & \in & \bar{mth}\ si\ \exists m(x)\ c_m \in \bar{mth}\ et\ c \in c_m \\ c & \in & \Sigma\ si\ \Sigma = (L, \sigma)\ et\ \exists i\ tel\ que\ c \in c^i \end{array}$$

Une variable est présente dans une instruction, noté $x \in c$, si la variable est utilisée dans

l'instruction.

$$\begin{array}{lcl} x & \in & x := e \\ x & \in & x := \texttt{allocate}(e) \\ x & \in & x := y[e] \\ x & \in & y := x[e] \\ x & \in & x[e_1] := e_2 \end{array}$$

On étend cette notion aux programmes.

Coloration Nous avons vu que l'on peut colorer un programme pour distinguer les instructions instrumentées. Les définitions suivants permettent de définir des programmes aux colorations cohérentes.

Définition 8.15 *Le prédicat $oneColor_{\square}(c)$ indique si une instruction c est colorée de manière uniforme avec la couleur $\square$.*

$$\begin{array}{rcl} oneColor_{\square}(\texttt{while } b_1 \texttt{ do } b) & = & oneColor_{\square}(b) \\ oneColor_{\square}(\texttt{if } b \texttt{ then } b_1 \texttt{ else } b_2) & = & oneColor_{\square}(b_1) \wedge oneColor_{\square}(b_2) \\ oneColor_{\square}(c) & & \\ oneColor_{\square}([]) & & \\ oneColor_{\square}(c; b) & = & oneColor_{\square}(b) \end{array}$$

Définition 8.16 *Un bloc d'instruction b est bien coloré, $goodColor(b)$ si i pour tout* $\texttt{while } b_1 \texttt{ do } b_1, \texttt{if } b \texttt{ then } b_1 \texttt{ else } b_2 \in b$, *on a* $oneColor_{\blacksquare}(\texttt{while } b_1 \texttt{ do } b_1)$ $oneColor_{\blacksquare}(\texttt{if } b \texttt{ then } b_1 \texttt{ else } b_2)$

Nous utilisons ce prédicat pour définir les programmes bien colorées.

Définition 8.17 *Un programme $\overline{mth}$ b est bien coloré si pour tout $b_m \in \overline{mth}$ on a $goodColor(b_m) \wedge goodColor(b)$*

Mesure d'un état Pour prouver l'équivalence d'état, nous avons besoin de savoir que nous n'avons pas affaire à une suite de réductions infinie dans les instructions instrumentées. C'est pourquoi nous avons besoin d'une mesure sur un état. Une partie de cette mesure prend en compte le nombre d'instructions restantes à exécuter.

Définition 8.18 *Nous définissons la taille d'un bloc d'instruction b, notée $|b|$. La taille représente le nombre d'instructions à exécuter.*

$$\begin{array}{rcl} |[]| & = & 0 \\ |\texttt{if then } b_1 \texttt{ else } b_2; b| & = & 1 + |b_1| + |b_2| + |b| \\ |c; b| & = & 1 + |b| \text{ où } c \neq \texttt{if then else} \end{array}$$

Nous pouvons maintenant définir la mesure sur un état.

Définition 8.19 *La mesure s'applique a un état et n'est définie que pour les états dont l'instruction en tête est une instruction instrumentée. Elle est définie en fonction de l'instruction*

en tête, et du bloc d'instructions auquel elle appartient. Nous avons une mesure pour le bloc removeList et une pour les autres blocs. La définition naturelle est le nombre d'instruction restantes à exécuter. La définition de la mesure est plus difficile lorsqu'un boucle est exécutée. En effet, dans ce cas, le nombre d'instructions fluctue. Nous ne pouvons donc pas nous baser sur le nombre d'instructions pour définir la mesure dans ce cas. Seul dans le bloc removeList est présente une boucle. Dans ce cas, nous définissons la mesure comme une combinaison du nombre d'instructions restantes, et la taille de la liste partant de la variable current. La mesure est un couple d'entier dans ce cas. Soit un état Σ et b le bloc d'instructions en tête de la pile d'instructions de Σ, alors ${n^0}_\Sigma$ est la taille de la liste partant de $arg[0]$ et n_Σ est la taille de la liste partant de la variable current.

- *removeList : La mesure dépend de la position pos dans* $instr_{removeList}(\Sigma, pos)$
  - $0 : ({n^0}_\Sigma, |b|)$
  - $1, 2, 3, 3.0.0, 3.1.0, 4 : (n_\Sigma, |b|)$
  - $2.0, 2.1 : (n_\Sigma - 1, |b|)$
- *autres blocs d'instructions instrumentées : la mesure correspond au nombre d'instructions restantes :* $|L|$

Définition 8.20 *L'ordre utilisé pour comparer les mesures sur les états est le suivant :*

$$\begin{array}{lll} (n_1, n_2) & <_{lex} & (n_3, n_4) \; si \; n_1 < n_2 \\ (n_1, n_2) & <_{lex} & (n_3, n_4) \; si \; n_1 = n_3 \wedge n_2 < n_4 \end{array}$$

Cet ordre est bien fondé car $<$ est bien fondé sur les entiers positifs.

Définition 8.21 *L'état initial d'un programme $\overline{mth}$ b est le suivant :* $(\overline{mth}, (b, \emptyset), \emptyset)$

8.4.2 Invariant

Afin d'établir l'équivalence, nous avons besoin de définir un invariant sur l'état d'un programme compilé. La définition de cet invariant nécessite l'introduction de définitions supplémentaires.

Nous disposons d'un prédicat qui pour tout bloc, indique si c'est un bloc d'instructions instrumentées.

$$\begin{array}{rcl} blockInstrumented(b) & = & b = initProgram \vee \\ & & \exists\, e,\ b = prepareCallMethod(e) \vee \\ & & \exists\, x,\ b = initMethod(x) \vee \\ & & \exists\, x, e,\ b = removeList(x, e) \end{array}$$

A chaque instruction de ces blocs, est associée une propriété qui est valide avant l'exécution de l'instruction. Elle est visible dans la troisième colonne des figures (8.4),(8.5),(8.6),(8.8), (8.9) et (8.10). Cette proposition est notée

$$prop_{bloc}(\Sigma, pos)$$

Définition 8.22 *Les prédicats suivant sont utilisés pour définir l'invariant d'un état. Ces prédicats seront utilisés en temps que garde, qui agissent en temps que contrôle de la position de l'instruction active.*

$$\texttt{Initialised}(\Sigma) \equiv \nexists pos, instrAt(initProgram, pos) = instrTop_\Sigma$$

$$\texttt{NodeAdded}(\Sigma) \equiv \nexists pos, instrAt(addList(\cdot,\cdot), pos) = instrTop_\Sigma$$

$$\texttt{NodeRemoved}(\Sigma) \equiv \nexists pos, instrAt(removeList(\cdot), pos) = instrTop_\Sigma$$

Afin d'améliorer la lisibilité, nous avons plusieurs constantes pour désigner les champs des enregistrements. Ces constantes débutent toutes par F_.

Nous notons les adresses stockées dans le champs F_ADD de la liste partant de $\sigma(\rho(arg))(0)$*:*

$$lista_{\sigma,\rho} = infoList_\sigma(\sigma(\rho(arg))(0))$$

Nous notons les adresses constituant les pointeurs de la liste partant de $\sigma(\rho(arg))(0)$ *sans l'adresse* `null` *:*

$$listdp_{\sigma,\rho} = nodeList_\sigma(\sigma(\rho(arg))(0))\backslash\texttt{null}$$

Nous pouvons maintenant définir l'invariant d'un état accessible Σ *depuis un programme compilé* $\mathcal{C}(p)$ *(noté* $\mathcal{C}(p) \hookrightarrow \Sigma$*):*

$$Inv(\Sigma) \equiv$$

$$\mathcal{C}(p) \hookrightarrow \Sigma = (\overline{mth}, L, \sigma) \wedge$$

$$\forall \rho \in L, ((\texttt{Initialised}(\Sigma) \Rightarrow \exists xs,\ Liste(\sigma, \sigma(\rho(arg))(0), xs)) \wedge \tag{8.2}$$

$$(\exists m,\ instrTop_\Sigma = m(\cdot) \Rightarrow \forall \rho_t,\ L = (\cdot, \rho_t)\cdot\_ \wedge \sigma(\rho(arg))(1) = \rho_t(mExpr)) \wedge \tag{8.3}$$

$$\texttt{NodeAdded}(\Sigma) \Rightarrow \texttt{NodeRemoved}(\Sigma) \Rightarrow \texttt{Initialised}(\Sigma) \Rightarrow \tag{8.4}$$
$$(listdp_{\sigma,\rho} \cup lista_{\sigma,\rho} \cup \rho(\text{arg}) = dom(\sigma)) \wedge$$

$$\texttt{Initialised}(\Sigma) \Rightarrow (listdp_{\sigma,\rho} \cap lista_{\sigma,\rho} \cap \rho(\text{arg}) = \emptyset) \wedge \tag{8.5}$$

$$(\forall bloc,\ blockInstrumented(bloc) \Rightarrow \tag{8.6}$$
$$\exists pos, instrAt(bloc, pos) = instrTop_\Sigma \Rightarrow prop_{bloc}(\Sigma, pos)))$$

Définition 8.23 *Les tableaux (figures 8.6, 8.5, 8.8, 8.9 et 8.10) définissent les propriétés valides aux différentes positions des différents blocs d'instructions instrumentées (block) définies sur un état* Σ *et notées* $prop_{block}(\Sigma, position)$*. Nous utilisons dans les propriétés un tas* σ *et un environnement local* ρ *tel que* $\Sigma = (\overline{mth}, L, \sigma)$*. Les propriétés sont exprimées pour tout environnement local* ρ *appartenant à* L*.*

Position	Instruction	Propriété
0	$arg := allocate(2)$	$\rho = \emptyset \wedge \sigma = \emptyset$
1	$arg[0] := \texttt{null}$	$\exists \ell,\ \sigma = \ell \mapsto [0,0] \wedge \rho = [arg \mapsto \ell]$
2	$arg[1] := 0$	$\exists \ell,\ \sigma = \ell \mapsto [\texttt{null}, 0] \wedge \rho = [arg \mapsto \ell]$

FIGURE 8.4 – *Préconditions pour chaque instruction de initProgram*

Position	Instruction	Propriété
0	mExpr:= e	*True*
1	arg[1] := mExpr	$\exists v,\ [\![\texttt{e}]\!] = v$

FIGURE 8.5 – *Préconditions pour chaque instruction de prepareCallMethod*(e)

8.4.3 Préservation sémantique

Nous souhaitons établir certains résultats intermédiaires afin de prouver la conservation de l'invariant et l'équivalence des états.

Étant donné une réduction d'un état d'un programme vers un autre état, on souhaite avoir certaines informations sur l'instruction active suivante, afin de savoir quelles gardes deviennent actives et donc comment prouver l'invariant.

Position	Instruction	Propriété
0	x := arg[1]	$L = (\cdot, \rho') \cdot (\cdot, \rho) \cdot \_ \wedge$ $\sigma(\rho'(arg))(1) = \rho(mExpr)$

FIGURE 8.6 – *Préconditions pour chaque instruction de initMethod*(x)

Position	Instruction	Propriété
0	node:= allocate(3);	$listdp_{\sigma,\rho} \cup lista_{\sigma,\rho} \cup \rho(arg) \cup \rho(\mathtt{x}) = dom(\sigma) \wedge$ $listdp_{\sigma,\rho} \cap lista_{\sigma,\rho} \cap \rho(arg) \cap \rho(\mathtt{x}) = \emptyset$
1	size:=e;	$\exists \ell, \rho(node) = \ell \wedge$ $\sigma(\ell) = [0,0,0] \wedge$ $\ell \notin nodeList_\sigma(\sigma(\rho(arg)(0))) \wedge$ $\rho(node) \neq \rho(arg) \wedge$ $listdp_{\sigma,\rho} \cup lista_{\sigma,\rho} \cup \rho(arg) \cup \rho(\mathtt{x}) \cup \rho(node) = dom(\sigma) \wedge$ $listdp_{\sigma,\rho} \cap lista_{\sigma,\rho} \cap \rho(arg) \cap \rho(\mathtt{x}) \cap \rho(node) = \emptyset$
2	node[F_ADD] := x	$\exists \ell, v_s, \rho(node) = \ell \wedge$ $\sigma(\ell) = [0,0,0] \wedge$ $\rho(size) = v_s \wedge$ $\ell \notin nodeList_\sigma(\sigma(\rho(arg)(0))) \wedge$ $\rho(node) \neq \rho(arg) \wedge$ $listdp_{\sigma,\rho} \cup lista_{\sigma,\rho} \cup \rho(arg) \cup \rho(\mathtt{x}) \cup \rho(node) = dom(\sigma) \wedge$ $listdp_{\sigma,\rho} \cap lista_{\sigma,\rho} \cap \rho(arg) \cap \rho(\mathtt{x}) \cap \rho(node) = \emptyset$
3	node[F_SIZE] := size;	$\exists \ell, v_s, v, \rho(node) = \ell \wedge$ $\rho(\mathtt{x}) = v \wedge$ $\sigma(\ell) = [v,0,0] \wedge$ $\rho(size) = v_s \wedge$ $\ell \notin nodeList_\sigma(\sigma(\rho(arg)(0))) \wedge$ $\rho(node) \neq \rho(arg) \wedge$ $listdp_{\sigma,\rho} \cup lista_{\sigma,\rho} \cup \rho(arg) \cup \rho(\mathtt{x}) \cup \rho(node) = dom(\sigma) \wedge$ $listdp_{\sigma,\rho} \cap lista_{\sigma,\rho} \cap \rho(arg) \cap \rho(\mathtt{x}) \cap \rho(node) = \emptyset$
3.5	tmp:= $arg[0]$	$\exists \ell, v_s, v, \rho(node) = \ell \wedge$ $\rho(\mathtt{x}) = v \wedge$ $\sigma(\ell) = [v, v_s, 0] \wedge$ $\rho(size) = v_s \wedge$ $\ell \notin nodeList_\sigma(\sigma(\rho(arg)(0))) \wedge$ $\rho(node) \neq \rho(arg) \wedge$ $listdp_{\sigma,\rho} \cup lista_{\sigma,\rho} \cup \rho(arg) \cup \rho(\mathtt{x}) \cup \rho(node) = dom(\sigma) \wedge$ $listdp_{\sigma,\rho} \cap lista_{\sigma,\rho} \cap \rho(arg) \cap \rho(\mathtt{x}) \cap \rho(node) = \emptyset$

FIGURE 8.7 – *Préconditions pour chaque instruction de addList*(x, e) – *partie* 1

Position	Instruction	Propriété
4	$node[F\_NEXT] := tmp$	$\exists \ell, v_s, v, \rho(node) = \ell \wedge$ $\rho(\mathtt{x}) = v \wedge$ $\sigma(\ell) = [v, v_s, 0] \wedge$ $\rho(size) = v_s \wedge$ $tmp = \ell_1 \wedge$ $\sigma(\rho(arg))(0) = \ell_1 \wedge$ $\ell \notin nodeList_\sigma(\sigma(\rho(arg)(0))) \wedge$ $\rho(node) \neq \rho(arg) \wedge$ $listdp_{\sigma,\rho} \cup lista_{\sigma,\rho} \cup \rho(arg) \cup \rho(\mathtt{x}) \cup \rho(node) = dom(\sigma) \wedge$ $listdp_{\sigma,\rho} \cap lista_{\sigma,\rho} \cap \rho(arg) \cap \rho(\mathtt{x}) \cap \rho(node) = \emptyset$
5	$arg[0] := node$	$\exists \ell, v_s, v, \rho(node) = \ell \wedge$ $\rho(\mathtt{x}) = v \wedge$ $\sigma(\ell) = [v, v_s, \ell_1] \wedge$ $\rho(size) = v_s \wedge$ $\sigma(\rho(arg))(0) = \ell_1 \wedge$ $\ell \notin nodeList_\sigma(\sigma(\rho(arg)(0))) \wedge$ $listdp_{\sigma,\rho} \cup lista_{\sigma,\rho} \cup \rho(arg) \cup \rho(\mathtt{x}) \cup \rho(node) = dom(\sigma) \wedge$ $listdp_{\sigma,\rho} \cap lista_{\sigma,\rho} \cap \rho(arg) \cap \rho(\mathtt{x}) \cap \rho(node) = \emptyset$

FIGURE 8.8 – *Préconditions pour chaque instruction de addList*$(\mathtt{x}, \mathtt{e})$ *– partie* 2

Lemme 8.7 *Étant donné deux états accessibles d'un programme* $\mathcal{C}(p)$ Σ *et* Σ'*, telle que* $\vdash_{\mathcal{C}(p)} \Sigma \rightarrow \Sigma'$*, le prédicat suivant met en relation les instructions instrTop de chacun des deux états :*

$(\exists pos, b,\ S\ pos < |b| \wedge blockInstrumented(b) \wedge instrAt(b, pos) = instrTop_\Sigma \wedge$
$instrAt(b, pos) \neq \mathtt{if} \cdot \mathtt{then} \cdot \mathtt{else} \cdot \wedge instrAt(b, pos) \neq \mathtt{while} \cdot \mathtt{do} \cdot \Rightarrow$
$\quad instrAt(b, S\ pos) = instrTop_{\Sigma'}) \wedge$

$(\exists pos, b,\ S\ pos = |b| \wedge blockInstrumented(b) \wedge \forall e,\ b \neq prepareCallMethod(e)$
$\wedge instrAt(b, pos) = instrTop_\Sigma \Rightarrow$
$\quad (instrTop_{\Sigma'} = \blacksquare \vee \exists e,\ instrTop_{\Sigma'} = fst(prepareCallMethod(e))) \wedge \forall m, instrTop_{\Sigma'} \neq m(\cdot)) \wedge$

$(\exists pos,\ b_0,\ b,\ \wedge instrTop_\Sigma = \mathtt{while} \cdot \mathtt{do}\ b|\blacksquare \wedge blockInstrumented(b_0) \wedge$
$instrAt(b_0, pos) = instrTop_\Sigma \Rightarrow$
$\quad instrTop_{\Sigma'} = fst(b) \vee instrAt(b, S\ pos) = instrTop_{\Sigma'}) \wedge$

$(b_1, b_2, \wedge \mathtt{if}\ e\ \mathtt{then}\ b_1\ \mathtt{else}\ b_2 = instrTop_\Sigma \Rightarrow$
$\quad \rho = top\ env\ \Sigma \wedge ((\llbracket e \rrbracket = true \wedge instrTop_{\Sigma'} = fst(b_1)) \vee (\llbracket e \rrbracket = false \wedge instrTop_{\Sigma'} = fst(b_2)))) \wedge$

$(\exists b,\ \wedge instrTop_\Sigma = \mathtt{while} \cdot \mathtt{do}\ b|\blacksquare \Rightarrow$
$\quad instrTop_{\Sigma'} = fst(b) \vee (\exists e,\ instrTop_{\Sigma'} = fst(prepareCallMethod(e))) \wedge \forall m, instrTop_{\Sigma'} \neq m(\cdot))) \wedge$

$(instrTop_\Sigma = \blacksquare \wedge instrTop_\Sigma \neq \mathtt{allocate}(\cdot) \wedge instrTop_\Sigma \neq \mathtt{dispose}(\cdot) \wedge$
$instrTop_\Sigma \neq \mathtt{if} \cdot \mathtt{then} \cdot \mathtt{else} \cdot \wedge instrTop_\Sigma \neq \mathtt{while} \cdot \mathtt{do} \cdot \wedge \forall m, instrTop_\Sigma \neq m(\cdot) \Rightarrow$
$\quad (instrTop_{\Sigma'} = \blacksquare \vee \exists e,\ instrTop_{\Sigma'} = fst(prepareCallMethod(e))) \wedge \forall m, instrTop_{\Sigma'} \neq m(\cdot)) \wedge$

$(instrTop_\Sigma = \mathtt{allocate}(\cdot) \Rightarrow \exists x,\ e,\ instrTop_{\Sigma'} = fst(addList(x, e))) \wedge$

$(instrTop_\Sigma = \mathtt{dispose}(\cdot) \Rightarrow \exists e,\ instrTop_{\Sigma'} = fst(removeList(e))) \wedge$

$(\exists m, instrTop_\Sigma = m(\cdot) \Rightarrow \exists x,\ instrTop_{\Sigma'} = fst(initMethod(x))) \wedge$

$(\exists e,\ instrTop_\Sigma = lst(prepareCallMethod(e)) \Rightarrow \exists m, instrTop_{\Sigma'} = m(\cdot))$

Position	Instruction	Propriété
0	$current := arg[0];$	$\exists \ell,\ [\![\mathtt{e}]\!] = \ell \wedge$ $listdp_{\sigma,\rho} \cup lista_{\sigma,\rho} \cup \rho(arg) = dom(\sigma) \cup \ell \wedge$ $listdp_{\sigma,\rho} \cap lista_{\sigma,\rho} \cap \rho(arg) = \emptyset \wedge$ $\rho(arg) \neq \mathtt{null} \wedge$ $\ell \in lista_{\sigma,\rho}$
1	$addrRm := \mathtt{e};$	$\exists \ell,\ [\![\mathtt{e}]\!] = \ell \wedge$ $\rho(current) = \sigma(\rho(arg))(0) \wedge$ $listdp_{\sigma,\rho} \cup lista_{\sigma,\rho} \cup \rho(arg) = dom(\sigma) \cup \ell \wedge$ $listdp_{\sigma,\rho} \cap lista_{\sigma,\rho} \cap \rho(arg) = \emptyset \wedge$ $\rho(arg) \neq \mathtt{null} \wedge$ $\rho(current) \neq \mathtt{null} \wedge$ $\ell \in infoList_{\sigma}(\rho(current))$
1.5	$tmp := current[F\_ADD];$	$\exists \ell,\ [\![\mathtt{e}]\!] = \ell \wedge$ $\exists xs, \exists xs',\ Liste(\sigma, \sigma(\rho(current))(0), xs') \wedge$ $Liste(\sigma, \sigma(\rho(arg))(0), xs) \wedge$ $\exists xs'' xs = xs'' + + xs' \wedge$ $(\rho(current) \neq \sigma(\rho(arg))(0) \rightarrow$ $\quad \sigma(\rho(oldCurrent))(F\_NEXT) = \rho(current)) \wedge$ $listdp_{\sigma,\rho} \cup lista_{\sigma,\rho} \cup \rho(arg) = dom(\sigma) \cup \ell \wedge$ $listdp_{\sigma,\rho} \cap lista_{\sigma,\rho} \cap \rho(arg) = \emptyset \wedge$ $\rho(arg) \neq \mathtt{null} \wedge$ $\rho(current) \neq \mathtt{null} \wedge$ $\rho(addrRm) = \ell \wedge$ $\ell \in infoList_{\sigma}(\rho(current))$
2	$while(\neg(tmp = addrRm))\{$ $oldCurrent := current;$ $current := current[F\_NEXT];$ $tmp := current[F\_ADD]$ $\}$	$\exists \ell,\ [\![\mathtt{e}]\!] = \ell \wedge$ $\exists xs, \exists xs',\ Liste(\sigma, \sigma(\rho(current))(0), xs') \wedge$ $Liste(\sigma, \sigma(\rho(arg))(0), xs) \wedge$ $\exists xs'' xs = xs'' + + xs' \wedge$ $(\rho(current) \neq \sigma(\rho(arg))(0) \rightarrow$ $\quad \sigma(\rho(oldCurrent))(F\_NEXT) = \rho(current)) \wedge$ $listdp_{\sigma,\rho} \cup lista_{\sigma,\rho} \cup \rho(arg) = dom(\sigma) \cup \ell \wedge$ $listdp_{\sigma,\rho} \cap lista_{\sigma,\rho} \cap \rho(arg) = \emptyset \wedge$ $\rho(arg) \neq \mathtt{null} \wedge$ $\rho(current) \neq \mathtt{null} \wedge$ $\rho(addrRm) = \ell \wedge$ $\ell \in infoList_{\sigma}(\rho(current)) \wedge$ $\rho(tmp) = \sigma(\rho(current))(F\_ADD)$
2.0	$oldCurrent := current;$	$\exists \ell,\ [\![\mathtt{e}]\!] == \ell \wedge$ $\exists xs, \exists xs' Liste(\sigma, \sigma(\rho(current))(0), xs') \wedge$ $Liste(\sigma, \sigma(\rho(arg))(0), xs) \wedge$ $\exists xs'' xs = xs'' + + xs' \wedge$ $(\rho(current) \neq \sigma(\rho(arg))(0) \rightarrow$ $\quad \sigma(\rho(oldCurrent))(F\_NEXT) = \rho(current)) \wedge$ $listdp_{\sigma,\rho} \cup lista_{\sigma,\rho} \cup \rho(arg) = dom(\sigma) \cup \ell \wedge$ $listdp_{\sigma,\rho} \cap lista_{\sigma,\rho} \cap \rho(arg) = \emptyset \wedge$ $\sigma(\rho(current))(F\_ADD) \neq \ell \wedge$ $\rho(arg) \neq \mathtt{null} \wedge$ $\rho(current) \neq \mathtt{null} \wedge$ $\rho(addrRm) = \ell \wedge$ $\ell \in infoList_{\sigma}(\rho(current))$

FIGURE 8.9 – *Préconditions pour chaque instruction de removeList*(e) – *partie* 1

Position	Instruction	Propriété
2.1	$current := current[F\_NEXT];$	$\exists \ell,\ [\![\mathtt{e}]\!] = \ell \wedge$ $\exists xs, \exists xs' Liste(\sigma, \rho(current), xs') \wedge$ $Liste(\sigma, \sigma(\rho(arg))(0), xs) \wedge$ $\exists xs'' xs = xs'' + + xs' \wedge$ $\rho(oldCurrent) = \rho(current) \wedge$ $listdp_{\sigma,\rho} \cup lista_{\sigma,\rho} \cup \rho(arg) = dom(\sigma) \cup \ell \wedge$ $listdp_{\sigma,\rho} \cap lista_{\sigma,\rho} \cap \rho(arg) = \emptyset \wedge$ $\sigma(\rho(current))(F\_ADD) \neq \ell \wedge$ $\rho(arg) \neq \mathtt{null} \wedge$ $\rho(current) \neq \mathtt{null} \wedge$ $\rho(addrRm) = \ell \wedge$ $\ell \in infoList_{\sigma}(\rho(current))$
2.2	$tmp := current[F\_ADD];$	$\exists \ell,\ [\![\mathtt{e}]\!] = \ell \wedge$ $\exists xs, \exists xs' Liste(\sigma, \sigma(\rho(current))(0), xs') \wedge$ $Liste(\sigma, \sigma(\rho(arg))(0), xs) \wedge$ $\exists xs'' xs = xs'' + + xs' \wedge$ $(\rho(current) \neq \sigma(\rho(arg))(0) \rightarrow$ $\quad \sigma(\rho(oldCurrent))(F\_NEXT) = \rho(current)) \wedge$ $listdp_{\sigma,\rho} \cup lista_{\sigma,\rho} \cup \rho(arg) = dom(\sigma) \cup \ell \wedge$ $listdp_{\sigma,\rho} \cap lista_{\sigma,\rho} \cap \rho(arg) = \emptyset \wedge$ $\rho(current) \neq \mathtt{null} \wedge$ $\rho(arg) \neq \mathtt{null} \wedge$ $\sigma(\rho(current))(F\_ADD) = \ell \wedge$ $(\rho(current) \neq \sigma(\rho(arg))(0) \rightarrow$ $\quad (\rho(oldCurrent)) \neq \mathtt{null})$
3	$if(tmp = current) then$ $tmp := current[F\_NEXT];$ $arg[0] := tmp;$ $else$ $tmp := current[F\_NEXT];$ $oldCurrent[F\_NEXT] := tmp$	$\exists \ell,\ [\![\mathtt{e}]\!] = \ell \wedge$ $\exists xs, \exists xs' Liste(\sigma, \sigma(\rho(current))(0), xs') \wedge$ $Liste(\sigma, \sigma(\rho(arg))(0), xs) \wedge$ $\exists xs'' xs = xs'' + + xs' \wedge$ $(\rho(current) \neq \sigma(\rho(arg))(0) \rightarrow$ $\quad \sigma(\rho(oldCurrent))(F\_NEXT) = \rho(current)) \wedge$ $listdp_{\sigma,\rho} \cup lista_{\sigma,\rho} \cup \rho(arg) = dom(\sigma) \cup \ell \wedge$ $listdp_{\sigma,\rho} \cap lista_{\sigma,\rho} \cap \rho(arg) = \emptyset \wedge$ $\rho(current) \neq \mathtt{null} \wedge$ $\rho(arg) \neq \mathtt{null} \wedge$ $\sigma(\rho(current))(F\_ADD) = \ell \wedge$ $(\rho(current) \neq \sigma(\rho(arg))(0) \rightarrow$ $\quad (\rho(oldCurrent)) \neq \mathtt{null}) \wedge$ $\rho(tmp) = \sigma(\rho(arg))(F\_ADD)$
3.0.0	$tmp := current[F\_NEXT];$	$\exists \ell,\ [\![\mathtt{e}]\!] = \ell \wedge$ $\exists xs, \exists xs' Liste(\sigma, \sigma(\rho(current))(0), xs') \wedge$ $Liste(\sigma, \sigma(\rho(arg))(0), xs) \wedge$ $\exists xs'' xs = xs'' + + xs' \wedge$ $(\sigma(\rho(arg))(0) = \rho(current)) \wedge$ $\rho(current) \neq \mathtt{null} \wedge$ $\rho(arg) \neq \mathtt{null} \wedge$ $\sigma(\rho(current))(F\_ADD) = \ell \wedge$ $listdp_{\sigma,\rho} \cup lista_{\sigma,\rho} \cup \rho(arg) = dom(\sigma) \cup \ell \wedge$ $\rho(tmp) = \sigma(\rho(arg))(F\_ADD)$

FIGURE 8.10 – *Préconditions pour chaque instruction de removeList*(e) – *partie* 2

Position	Instruction	Propriété
3.0.1	$arg[0] := tmp;$	$\exists \ell,\ [\![\mathtt{e}]\!] = \ell \wedge$ $\exists xs, \exists xs' Liste(\sigma, \sigma(\rho(current))(0), xs') \wedge$ $Liste(\sigma, \sigma(\rho(arg))(0), xs) \wedge$ $\exists xs'' xs = xs'' + + xs' \wedge$ $(\sigma(\rho(arg))(0) = \rho(current)) \wedge$ $\rho(current) \neq \mathtt{null} \wedge$ $\rho(arg) \neq \mathtt{null} \wedge$ $\sigma(\rho(current))(F\_ADD) = \ell \wedge$ $listdp_{\sigma,\rho} \cup lista_{\sigma,\rho} \cup \rho(arg) = dom(\sigma) \cup \ell \wedge$ $\rho(tmp) = \sigma(\rho(arg))(F\_NEXT)$
3.1.0	$tmp := current[F\_NEXT];$	$\exists \ell,\ [\![\mathtt{e}]\!] = \ell \wedge$ $\exists xs, \exists xs' Liste(\sigma, \sigma(\rho(current))(0), xs') \wedge$ $Liste(\sigma, \sigma(\rho(arg))(0), xs) \wedge$ $\exists xs'' xs = xs'' + + xs' \wedge$ $\sigma(\rho(oldCurrent))(F\_NEXT) = \rho(current) \wedge$ $\rho(current) \neq \mathtt{null} \wedge$ $\rho(oldCurrent) \neq \mathtt{null} \wedge$ $\sigma(\rho(current))(F\_ADD) = \ell \wedge$ $listdp_{\sigma,\rho} \cup lista_{\sigma,\rho} \cup \rho(arg) = dom(\sigma) \cup \ell \wedge$
3.1.1	$oldCurrent[F\_NEXT] := tmp;$	$\exists \ell,\ [\![\mathtt{e}]\!] = \ell \wedge$ $\exists xs, \exists xs' Liste(\sigma, \sigma(\rho(current))(0), xs') \wedge$ $Liste(\sigma, \sigma(\rho(arg))(0), xs) \wedge$ $\exists xs'' xs = xs'' + + xs' \wedge$ $\sigma(\rho(oldCurrent))(F\_NEXT) = \rho(current) \wedge$ $\rho(current) \neq \mathtt{null} \wedge$ $\rho(oldCurrent) \neq \mathtt{null} \wedge$ $\sigma(\rho(current))(F\_ADD) = \ell \wedge$ $listdp_{\sigma,\rho} \cup lista_{\sigma,\rho} \cup \rho(arg) = dom(\sigma) \cup \ell \wedge$ $\rho(tmp) = \sigma(\rho(arg))(F\_NEXT)$
4	$dispose(current)$	$\rho(current) \neq \mathtt{null}$ $listdp_{\sigma,\rho} \cup lista_{\sigma,\rho} \cup \rho(arg) \cup \rho(current) = dom(\sigma) \wedge$ $listdp_{\sigma,\rho} \cap lista_{\sigma,\rho} \cap \rho(arg) \cap \rho(current) = \emptyset$

FIGURE 8.11 – *Préconditions pour chaque instruction de removeList*(e) *– partie* 3

Esquisse de preuve. On peut prouver le lemme 6.1 pour le même prédicat *executedCodeOf* pour notre langage impératif. Ceci nous permet de caractériser les positions de deux instructions qui se suivent dans le cas d'origine, tout au long de l'exécution. D'autre part la prédicat est vrai pour un programme compilé. Ces deux résultats permettent de conclure que le résultat est vrai pour tout le code accessible par une exécution du programme. □

Étant donné deux états équivalents, une expression s'évalue-t-elle de façon équivalente dans les deux états ?

Lemme 8.8 *Pour tout état $\Sigma_T\ \Sigma_S$, si $\Sigma_T \sim \Sigma_S$, pour tout environnement ρ_T, ρ_S appartenant à ces états et de même niveau, et soit σ_T, σ_S les tas des états, alors pour toute expression $e \in instrTop_{\Sigma_T} \wedge e_i ninstrTop_{\Sigma_S}$, nous avons soit*

— $\exists \ell_S, \ell_T, [\![ee]\!]_{\rho_S,\sigma_S} = \ell_S, [\![ee]\!]_{\rho_T,\sigma_T} = \ell_T$ *et* $\ell_S\ _{\rho_S,\sigma_S,\emptyset} \sim^{add}_{\rho_T,\sigma_T,\emptyset} \ell_T$,

— $[\![ee]\!]_{\rho_S,\sigma_S} = [\![ee]\!]_{\rho_T,\sigma_S}$ *sinon.*

Démonstration.

— Si l'expression est une constante, le résultat est immédiat

— Si l'expression est une variable

  — et que celle-ci contient une constante non adresse, l'équivalence des environnements locaux permet de conclure.

  — et que celle-ci contient une adresse, l'équivalence du tas permet de conclure

— Si l'expression contient une opération: L'opération ne peut pas porter sur les adresses car il n'y a pas d'arithmétique de pointeurs.

  — opération sur les entiers : on conclut avec une combinaison des arguments précédents

  — $[\![e_1 = e_2]\!]$: Suivant le résultat de l'évaluation :

    — true : On sait par définition de l'égalité sur les adresses que $[\![e_1]\!] = [\![e_2]\!] \wedge [\![e_1]\!] \in dom(\sigma) \wedge [\![e_2]\!] \in dom(\sigma)$. Soit $\ell = [\![e_1]\!]$. Nous considérons deux cas, suivant que cette adresse soit dans la liste d'exclusion $listp_{\sigma_T,\rho_T}$ ou non.

      — $\ell \notin listp_{\sigma_T,\rho_T}$: On sait par le lemme 8.3 que $G^{listp_{\sigma_T,\rho_T}}_{\sigma_T}([\![e_1]\!]) = G^{listp_{\sigma_T,\rho_T}}_{\sigma_T}([\![e_2]\!])$ et $G^{listp_{\sigma_T,\rho_T}}_{\sigma_T}([\![e_1]\!]) \neq \bot$. On sait que comme l'expression s'évalue en une adresse, l'expression est nécessairement une variable, car il n'y a pas de constante adresse. Par définition de l'équivalence on sait donc que $G^{listp_{\sigma_T,\rho_T}}_{\sigma_T}([\![e_1]\!]) = G^{\emptyset}_{\sigma_S}([\![e_1]\!])$ et $G^{listp_{\sigma_T,\rho_T}}_{\sigma_T}([\![e_2]\!]) = G^{\emptyset}_{\sigma_S}([\![e_2]\!])$. Comme l'on sait que $G^{\emptyset}_{\sigma_S}([\![e_1]\!]) \neq \bot$, en appliquant le lemme 8.4, on sait qu'il existe une adresse logique ℓ^p_1 telle que $\ell^p_1 \in dom(\sigma_S) \wedge G^{\emptyset}_{\rho_S}([\![e_1]\!]) = \ell^p_1$. Nous avons un résultat similaire pour $G^{\emptyset}_{\sigma_S}([\![e_2]\!])$. Avec le lemme 8.3 on sait que $\ell_1 = \ell_2$, et donc $[\![e_1 = e_2]\!] = true$. On a donc bien le même résultat pour l'évaluation.

      — $\ell \in listp_{\sigma_T,\rho_T}$: On sait donc par définition du graphe atteignable que $G^{listp_{\sigma_T,\rho_T}}_{\sigma_T}([\![e_1]\!]) = \bot$. Donc par équivalence des graphes, on

sait également que $G^{\emptyset}_{\sigma_S}([\![e_1]\!]) = \bot$. Par le lemme 8.5 on sait que $[\![e_1]\!] \notin dom(\sigma_S) \vee [\![e_1]\!] \in \emptyset$. On sait donc que $[\![e_1]\!] \notin dom(\sigma_S)$. Donc l'évaluation de l'expression n'est pas définie, le programme source est bloqué dans un état non final, ce qui est en contradiction avec l'hypothèse $safe(p)$, l'hypothèse $\ell \in listp_{\sigma_T,\rho_T}$ n'est pas possible.

— false : La preuve est similaire au cas précédent.

— non définie : On sait que $[\![e_1]\!] \notin dom(\sigma_T)$ ou $[\![e_2]\!] \notin dom(\sigma_T)$ ou les deux. Supposons que $[\![e_1]\!] \notin dom(\sigma_T)$ (le raisonnement est identique pour les autres cas). On sait donc que par définition du graphe atteignable que $G^{listp_{\sigma_T,\rho_T}}_{\sigma_T}([\![e_1]\!]) = \bot$. On sait aussi par équivalence des graphes que $G^{\emptyset}_{\sigma_S}([\![e_1]\!]) = \bot$. Donc par le lemme 8.5 on sait que $[\![e_1]\!] \notin dom(\sigma_S)$. Donc l'évaluation de l'expression n'est pas définie, le programme source est bloqué dans un état non final, ce qui est en contradiction avec l'hypothèse $safe(p)$. L'hypothèse de l'évaluation de l'égalité non définie n'est pas possible.

□

Lemme 8.9 *Pour tout programme p et état $(\overline{mth}, L, \sigma)$ tel que $\mathcal{C}(p) \hookrightarrow (\overline{mth}, L, \sigma)$ et pour toute instruction c qui est en tête de L,*

— $c = \blacksquare$, $\forall ci \in$ *code instrumenté*, $\neg suffixPrefixe(ci, c)$

— $c = \blacksquare$, $\exists! ci \in$ *code instrumenté*, $suffixPrefixe(ci, c)$

Esquisse de preuve. D'après la compilation, seules les instructions de bloc instrumentée sont de couleur ■. □

Équivalence

Lemme 8.10 *Pour tout bloc d'instruction b_1, b_2, instructions c_1, c_2 tas σ_1, σ_2 et environnements locaux ρ_1, ρ_2,si $d(c_1|\blacksquare; b_1) \sim^{\sigma_1\sigma_2}_{\rho_1\rho_2} c_2|\blacksquare; b_2$ alors $d(b_1) \sim^{\sigma_1\sigma_2}_{\rho_1\rho_2} b_2$.*

Esquisse de preuve. La preuve se fait par induction sur b_1. □

Lemme 8.11 *Pour tout bloc d'instruction b_1, b_2, instructions c tas σ_1, σ_2 et environnements locaux ρ_1, ρ_2, tel que $goodColor(c|\blacksquare; b_1) \wedge d(c|\blacksquare; b_1) \sim^{\sigma_1\sigma_2}_{\rho_1\rho_2} b_2$ alors $d(b_1) \sim^{\sigma_1\sigma_2}_{\rho_1\rho_2} b_2$.*

Esquisse de preuve. La preuve se fait par induction sur b_1. □

Lemme 8.12 *Pour tout bloc d'instruction b_1, b_2, b_{t1}, b_{f1}, b_{t2}, b_{f2},tas σ_1, σ_2 et environnements locaux ρ_1, ρ_2, tel que $d(\texttt{if then } b_{t1} \texttt{ else } b_{f1}|\blacksquare; b_1) \sim^{\sigma_1\sigma_2}_{\rho_1\rho_2} \texttt{if then } b_{t2} \texttt{ else } b_{f2}|\blacksquare; b_2$ alors $d(b_{t1}; b_1) \sim^{\sigma_1\sigma_2}_{\rho_1\rho_2} b_{t2}; b_2 \wedge d(b_{f1}; b_1) \sim^{\sigma_1\sigma_2}_{\rho_1\rho_2} b_{f2}; b_2$.*

Esquisse de preuve. La preuve se fait sur la définition de $d()$ et de l'équivalence $\sim$. □

Lemme 8.13 *Pour tout bloc d'instruction b_1, b2, b_{w1}, b_{w2}, tas σ_1, σ_2 et environnements locaux ρ_1, ρ_2, tel que $d(\texttt{while do } b_{w1}|\blacksquare; b_1) \sim^{\sigma_1\sigma_2}_{\rho_1\rho_2} \texttt{while do } b_{w2}|\blacksquare; b_2$ alors*

$$d(b_{w1}; \texttt{while do } b_{w1}; b_1) \sim^{\sigma_1\sigma_2}_{\rho_1\rho_2} b_{w2}; \texttt{while do } b_{w2}; b_2.$$

Esquisse de preuve. La preuve se fait sur la définition de $d()$. □

Lemme 8.14 *Pour tout programme $p = \overline{mth}\ b_{main}$, alors $b_{main} \sim^{\emptyset\emptyset}_{\emptyset\emptyset} \mathcal{C}(b_{main})$ et pour tout $b_m \in \overline{mth}$, $b_m \sim^{\emptyset\emptyset}_{\emptyset\emptyset} \mathcal{C}(b_m)$.*

Esquisse de preuve. Par induction sur b_m et par définition de la compilation. □

Lemme 8.15 *Pour tout programme p, et état initial de ce programme Σ^ϵ_S alors soit Σ^ϵ_T l'état initial du programme compilé $\mathcal{C}(p)$ on a $\Sigma^\epsilon_S \sim \Sigma^\epsilon_T$.*

Esquisse de preuve.

— $L_S \sim^{\text{instr}} L_T$: La pile d'instruction est de taille 1, et ne contient que le bloc d'instruction du programme principal. D'après le lemme 8.14, les deux bloc sont équivalents.

— $L_S \sim^{\text{env}} L_T$: La pile d'instruction est de taille 1, et l'environnement local est vide. On a donc l'équivalence.

— $\rho_T, \sigma_T \sim^{\text{heap}} \rho_S, \sigma_S$: La pile d'instruction est de taille 1, l'environnement local et le tas sont vides. On a donc l'équivalence.

□

Lemme 8.16 *Pour tout programme p et soit l'état initial du programme compilé Σ^ϵ_T, alors on a $Inv(\Sigma^\epsilon_T)$.*

Esquisse de preuve. La condition $\texttt{Initialised}(\Sigma^\epsilon_T)$ n'est pas vérifiée les invariants 8.2, , 8.4, 8.5 sont donc valides. La condition $\exists m,\ instrTop_{\Sigma^\epsilon_T} = m(\cdot)$ n'est pas vérifiée, l'invariant 8.3 est donc valide. La condition $prop_{initProgram}(\Sigma^\epsilon_T, 0)$ est vérifiée, les conditions $\rho = \emptyset\ \sigma = \emptyset$ sont valides par définition de Σ^ϵ_T (définition 8.21), et donc l'invariant 8.6 aussi. □

De manière identique aux instructions, nous ajoutons des couleurs aux actions pour distinguer celles issues d'instructions instrumentées ou non. L'action a la même couleur que l'instruction qui l'a générée.

Proposition 8.1 *Étant donné deux états Σ_T et Σ_S et un programme source p_S et le programme compilé $\mathcal{C}(p)$, et une action a tel que safe(p), $\Sigma_T \sim \Sigma_S$, $Inv(\Sigma_T)$ et $\vdash_{\mathcal{C}(p)} \Sigma_T \xrightarrow{a} \Sigma'_T$ alors :*

— *$Inv(\Sigma'_T)$*

— *et soit*

  — *L'action a n'est pas colorée, et il existe Σ'_S tel que $\vdash_p \Sigma_S \xrightarrow{a} \Sigma'_S$ et $\Sigma'_T \sim \Sigma'_S$*

— *L'action a est colorée et l'on a* $|\Sigma_T^{'}| < |\Sigma_T|$ *et* $\Sigma_T^{'} \sim \Sigma_S$

Esquisse de preuve. La preuve se fait sur par cas sur la règle de transition entre Σ_T et $\Sigma_T^{'}$. Pour chacun des cas nous inspectons si la couleur est ■ ou ■, afin de prouver la conservation de l'invariant et l'équivalence des états.

Considérons dans un premier temps les règles qui ne modifie pas la taille de la pile d'instructions. La preuve d'équivalence des instructions est identiques dans chacun des cas. Dans les cas ■, l'instruction est identique. Dans le cas ■ la définition d'équivalence d'instruction permet de conclure.

Dans un premier temps nous traitons le cas où la transition est de couleur ■.

Nous prouvons pour chacune des règles de la sémantique la conservation de l'invariant dans le nouvel état, et l'équivalence des nouveaux états sources $\Sigma_S^{'}$ et cibles $\Sigma_T^{'}$.

Pour l'équivalence des bloc d'instructions, $\Sigma_T^{'} \sim^{\text{instr}} \Sigma_S^{'}$, le lemme 8.10 permet de conclure dans chacun des cas.

— (**assign**) $x := e$: Le tas n'est pas modifié, et aucune variable réservée n'est modifiée, l'invariant n'est pas modifié. On sait grâce au lemme 8.8 que l'expression s'évalue de manière identique. Si l'expression s'évalue en une valeur non adresse, cette valeur est identique dans les deux cas, et donc l'environnement local est modifié de façon identique dans le programme source et compilé. Si l'expression s'évalue en une adresse, ces deux adresse sont équivalentes, les graphes mémoires restent donc équivalents.

— (**alloc**) `allocate`(e) : D'après le lemme 8.7, nous savons que l'instruction en tête de $\Sigma_T^{'}$ est $fst(addList(e))$ .Donc nous en déduisons que la propriété est $prop_{addList}(\Sigma_T^{'}, 0)$, soit $listdp_{\sigma,\rho} \cup lista_{\sigma,\rho} \cup \rho(arg) \cup \rho(\mathtt{x}) = dom(\sigma) \wedge listdp_{\sigma,\rho} \cap lista_{\sigma,\rho} \cap \rho(arg) \cap \rho(\mathtt{x}) = \emptyset$. Nous savons que $\rho(\mathtt{x})$ n'est pas dans le domaine du tas avant l'instruction. Nous déduisons donc cette propriété de l'invariant. L'équivalence des tas est prouvée à l'aide de lemmes 8.1 et 8.2.

— (**dispose**) `dispose`(e) : Pour la preuve de la propriété 8.2 de l'invariant, nous la prouvons par contradiction, en supposant que l'adresse supprimée est un des pointeurs de la liste de $arg[0]$. Dans ce cas la règle ne peut pas s'appliquer dans le programme source, car l'adresse n'est pas définie suivant l'équivalence des tas. Nous prouvons également la propriété $prop_{removeList(e)}(\Sigma_T^{'}, 0)$. On conclut sur l'équivalence des tas grâce aux lemmes 8.8et 8.2.

— (**get**)$x := y[e]$ La variable modifiée n'est pas réservée, donc cela n'a pas d'impact sur l'invariant. Seul l'environnement local est modifiée, et on sait grâce au lemme 8.8 que l'expression s'évalue de façon identique dans les deux programmes, et l'équivalence des tas permet de conclure.

— (**put**) $x[e] := y$: La preuve de la propriété 8.2 de l'invariant se fait par contradiction. En supposant que $x[e]$ fait référence à un élément de la liste de $arg[0]$, on prouve grâce à l'équivalence de tas, que la règle (**put**) ne peut pas s'appliquer dans le programme source, et qu'il est donc bloqué dans un état non final, ce qui est en contradiction avec l'hypothèse $safe(p)$. $x[e]$ n'est pas un élément de

la liste, la liste n'est donc pas modifiée. Pour les mêmes raisons, on sait que la liste de $arg[0]$ n'est pas modifiée, donc les propriétés 8.4 et 8.5 sont toujours valides. L'équivalence de tas est conservé car on sait que l'expression s'évalue de façon équivalente dans les deux programmes.

— **methodcall** $m(x)$: D'après le lemme 8.7, l'instruction suivante est $instrAt(initMethod, 0)$. La propriété à prouver est $prop_{initMethod}(\Sigma_T^{'}, 0) = L = (\cdot, \rho_T') \cdot (\cdot, \rho_T) \cdot \_ \wedge \sigma_T(\rho_T'(arg))(1) = \rho_T(mExpr)$. On sait d'après l'invariant 8.3 que $\sigma_T(\rho_T(arg))(1) = \rho_T(mExpr)$ où ρ_T est l'environnement local avant l'éxécution de l'instruction. De la règle **methodcall**, on sait que le nouvel environnement local en tête de pile (ρ_T') contient la variable arg, où $\rho_T'(arg) = \rho_T(arg)$. Cela permet de conclure. $\Sigma_T^{'} \sim \Sigma_S^{'}$:

  — $\Sigma_T^{'} \sim^{\text{instr}} \Sigma_S^{'}$: D'après la règle **methodcall**, on sait qu'un nouveau bloc d'instruction est ajouté en tête de pile, et le deuxième bloc est modifié. L'équivalence d'instruction pour le deuxième bloc est prouvée grâce au lemme 8.10. D'après la définition de la compilation, le nouveau bloc d'instruction est celui de la méthode, soit b_m, pour l'état $\Sigma_S^{'}$, et le bloc de la méthode compilé, pour l'état $\Sigma_T^{'}$, soit $initMethod; \mathcal{C}(b_m)$. D'après le lemme 8.14 ces deux blocs sont équivalents.

  — $\Sigma_T^{'} \sim^{\text{env}} \Sigma_S^{'}$: On sait d'après le lemme 8.7 que $\exists x,\ instrTop_{L_T'} = fst(initMethod(x))$. Il faut donc prouver que $\exists v, \ell,\ \rho_S^i = [x \mapsto v] \wedge \rho_T^i = [arg \mapsto \ell] \wedge \sigma(\rho_T^i(arg)(1)) = v$. On sait d'après l'équivalence $\Sigma_T \sim^{\text{instr}} \Sigma_S$ qu'en tête de pile de l'état source, il y a un appel de méthode. D'après la règle **methodcall**, on sait que dans le nouvel état source, le nouvel environnement ne contient qu'une variable. On a donc bien $\exists v, \ell,\ \rho_S^i = [x \mapsto v]$, et $[\![e]\!]$ où e est l'argument de la méthode. On sait aussi d'après l'équivalence d'instructions $\Sigma_T \sim^{\text{instr}} \Sigma_S$ que $m(mExpr) \sim^{m(}_{\rho_T \rho_S} e)$ et donc que $\rho_T mExpr = [\![e]\!]$. On sait grâce à l'invariant 8.3, que $\sigma(\rho_T(arg))(1) = \rho_T(mExpr)$. On a donc bien $\rho_T^i = [arg \mapsto \ell] \wedge \sigma(\rho_T^i(arg)(1)) = v)$.

  — $\Sigma_T^{'} \sim^{\text{heap}} \Sigma_S^{'}$: On sait d'après le lemme 8.7 que $\exists pos, x\ instrTop_{\rho_T} = instrAt(initMethod(x), pos)$ est vérifiée, il faut donc prouver que $reachableGraph_{\sigma_T}(\sigma(\rho_T(arg))(1), nodeList_\sigma(\sigma(\rho_T(arg))(0))) = reachableGraph_{\sigma_S}(\rho_S, \emptyset)$. Comme on l'a prouvé au précédent point on sait que $\exists \ell, \rho_T' = [arg \mapsto \ell] \wedge \rho_S' = [x \mapsto v] \wedge \sigma(\rho_T'(arg)(1)) = v$. Comme on a $\Sigma_T \sim^{\text{env}} \Sigma_S$, on a donc bien $reachableGraph_{\sigma_T}(\sigma(\rho_T(arg))(1), nodeList_\sigma(\sigma(\rho_T(arg))(0))) = reachableGraph_{\sigma_S}(\rho_S, \emptyset)$.

— **endmethod** : Le tas n'est pas modifié, un élément de la pile est supprimé. L'invariant et les équivalences sont conservés.

— **loop$_t$**, **loop$_f$**, **cond$_t$**, **cond$_f$** : Le tas et l'environnement local ne sont pas modifiés. Les invariants restent valides. Les lemmes 8.12, 8.13, et 8.11 permettent de conclure sur $\Sigma_T^{'} \sim^{\text{instr}} \Sigma_S^{'}$

La transition peut également être de couleur ■. Dans ce cas, d'après le lemme 8.9, l'instruction provient nécessairement d'un bloc d'instructions instrumentées. Après

chaque instruction, il est nécessaire de prouver que la mesure (définition 8.19) sur l'état décroît (définition 8.20) strictement afin d'éviter que des boucles se répètent indéfiniment. D'après la définition de la mesure, pour toutes les instructions en dehors du bloc *removeList*, celle-ci est définie par le nombre d'instruction restante. Comme il n'y a aucune boucle, la preuve de décroissance stricte est immédiate. La preuve sera donc détaillée uniquement pour les instructions du bloc *removeList*.

La preuve de $\Sigma_T^{'} \sim^{\text{instr}} \Sigma_S$ se fait par l'application du lemme 8.11.

— *initProgram* : Instruction position :
  — 0 : $arg := allocate(2)$: La preuve de l'invariant est immédiat car les gardes ne sont pas vérifiées pour la plupart des propriétés. La variable arg est réservée,donc cela ne modifie pas l'équivalence des environnements locaux. D'après la propriété, l'environnement local ne comporte que la variable arg. L'ensemble des variables racines $rootsVariables(\rho'_T)$ est donc vide, ainsi que le graphe mémoire.
  — 1: $arg[0] :=$ `null` : La preuve est identique à celle de l'instruction 0
  — 2: $arg[1] := 0$: D'après l'invariant 8.6, on sait que la liste partant de $arg[0]$ est vide, et l'on peut déduire les propriétés de l'invariant. La preuve de l'équivalence des états est identique à celle de l'instruction 0.

— *prepareCallMethod*(`e`)
  — 0 : $mExpr :=$ `e` : Le tas n'est pas modifié et $mExpr$ est une variable réservée, donc l'invariant, l'équivalence de tas et d'environnement est facilement prouvable. Pour l'équivalence d'instruction, $\Sigma_T^{'} \sim^{\text{instr}} \Sigma_S$, on sait que $hd(L_S) = m(\texttt{e})$. D'après la définition de la relation d'équivalence d'instruction et d'après le lemme 8.7, on sait que $hd(d(L'_T)) = m(mExpr)$. On sait également que $hd(d(L_T)) = m(\texttt{e})$. On sait que $\rho'_T(mExpr) = \rho'_T(\texttt{e})$, et donc que $[\![\texttt{e}]\!] = \rho'_T(mExpr)$ grâce à l'équivalence des environnement. La relation d'équivalence est valide.
  — 1 : $arg[1] := mExpr$: D'après le lemme 8.7, la condition $\exists m,\ instrTop_{\Sigma'_T} = m(\cdot)$ est vérifiée. L'instruction et la règle **put** permet de conclure sur la validité de la condition $\sigma(\rho(arg))(1) = \rho(mExpr)$ où ρ est l'environnement en tête de la pile. Le tas est modifié via la variable arg. On sait que l'adresse contenue dans arg est différente de celle de la liste partant de $arg[0]$ grâce à l'invariant 8.4. Cette écriture ne modifie donc pas l'invariant. L'adresse contenue dans la variable arg fait partie de la liste des adresses exclues du calcul du graphe atteignable. La modification de $arg[1]$ ne modifie donc pas le graphe, et conserve donc l'équivalence de tas. La preuve de l'équivalence d'instruction se base sur la propriété $prop_{prepareCallMethod(e)}(\Sigma_T^{'}, 1)$, et est similaire à l'équivalence d'instruction précédente.

— *initMethod*(`x`) Instruction position :
  — 0 : `x` := $arg[1]$: Les différentes propriétés de l'invariant sont valides car ni le tas, ni la variable arg ne sont modifiés. L'équivalence d'environnement locaux $\Sigma_T^{'} \sim^{\text{env}} \Sigma_S$ est valide car d'après l'invariant 8.3, on connaît le contenu

de $arg[1]$. L'affectation à la variable x permet d'établir l'équivalence des environnements.

— *addList*(x, e)
  — 0 : $node := allocate(3)$: La preuve de l'invariant se fait à l'aide de la règle d'allocation (**alloc**), qui ne modifie pas les valeurs des adresses existantes. La preuve d'équivalence des tas $\Sigma_T^{'} \sim^{heap} \Sigma_S$: La condition $\exists pos, x, e\ instrTop_{\rho_T} = instrAt(addList(x,e), pos) \wedge pos > 0$ est vérifiée, la propriété à prouver est donc $reachableGraph_{\sigma_T}(\rho_T, nodeList_\sigma(\sigma(\rho_T(arg))(0)) \cup \rho_T(node)) = reachableGraph_{\sigma_S}(\rho_S, \emptyset)$. $node$ est une variable réservée, donc elle n'est pas prise en compte dans le calcul du graphe. L'adresse allouée, est contenue dans $node$, qui fait partie de la liste d'exclusion. Si cette adresse était présente dans le tas, elle était associée à $\perp$ dans le calcul du graphe, car l'adresse n'était pas allouée. Dans le nouveau graphe, comme l'adresse fait partie de la liste d'exclusion, l'adresse est aussi associée à $\perp$, l'équivalence est donc valide.
  — 1 : $size$:= e : Le tas n'est pas modifié, ni la variable arg, ce qui conserve l'invariant, et l'équivalence des états.
  — 2 : $node[F\_ADD]$:= x : On sait d'après l'invariant que modifier le contenu de la variable $node$ ne modifie pas la liste de arg.
  — 3 : $node[F\_SIZE]$:= $size$: L'instruction est similaire à l'instruction 2, la preuve est donc similaire.
  — 3.5 : $tmp := node[F\_NEXT]$: L'instruction est similaire à l'instruction 1, la preuve des invariants différent de 8.6 est donc identique à la preuve de l'instruction 2.
  — 4 : $node[F\_NEXT] := tmp$: L'instruction est similaire à l'instruction 2, la preuve est donc similaire.
  — 5 : $arg[0] := node$ En inspectant la propriété $prop_{bloc}(\Sigma, 5)$, on voit que l'élément à l'adresse contenue dans $node$ est bien une liste, il y a donc toujours bien une liste dans $arg[0]$. On sait d'après la propriété $prop_{addList}(\Sigma, 5)$ et l'instruction que dans le nouvel état, $\rho(node) \in listp_{\sigma',\rho'}$. On sait aussi que $\rho(x) = \sigma(\rho(node))(F\_ADD)$, et donc dans le nouvel état on a $\rho(x) \in lista_{\sigma',\rho'}$. Équivalence $\Sigma_T^{'} \sim \Sigma_S$: Pour l'équivalence des tas, $\Sigma_T^{'} \sim^{heap} \Sigma_S$, on sait que $reachableGraph_{\sigma_T}(\rho_T, listp_{\sigma,\rho} \cup \rho_T(node)) = reachableGraph_{\sigma_S}(\rho_S, \emptyset)$. On veut prouver $reachableGraph_{\sigma_T}(\rho_T, listp_{\sigma,\rho}) = reachableGraph_{\sigma_S}(\rho_S, \emptyset)$. D'après la l'invariant 8.6 on sait que $\sigma(\rho(node))(F\_NEXT) = \sigma(\rho(arg))(0)$ avant l'instruction. Donc $listp_{\sigma,\rho} \cup \rho_T(node)$ avant l'instruction est identique à $listp_{\sigma,\rho}$ après l'instruction. L'invariant est donc vérifié.

— *removeList*(e) La propriété $\exists \ell, [\![e]\!] = \ell$ est présente pour chaque instruction. L'expression provient du programme source, donc son évaluation n'est pas modifiée par la modification de variable instrumentée.
  — 0 : $current := arg[0]$: Le tas n'est pas modifié, ni la variable arg. On déduit l'invariant de la propriété et de la règle d'affectation (**assign**). Pour l'équivalence de tas $\Sigma_T^{'} \sim^{heap} \Sigma_S$, la condition

$\exists pos, x, e$ $instrTop_{\rho_T} = instrAt(removeList(e), pos)$ est vérifiée, la propriété à prouver est donc $reachableGraph_{\sigma_T}(\rho_T, nodeList_\sigma(\sigma(\rho_T(arg))(0)) \cup \rho_T(current)) = reachableGraph_{\sigma_S}(\rho_S, \emptyset)$. C'est la propriété à prouver pour les instructions du bloc. Le tas n'est pas modifié, on déduit la propriété de $\Sigma_T^{'} \sim^{heap} \Sigma_S$. La mesure de Σ_T est ${n^0}_{\Sigma_T};7$. La mesure de $\Sigma_T^{'}$ est $n_{\Sigma_T^{'}};6$ du fait du nombre d'instruction restante dans le bloc. Pour rappel,${n^0}_\Sigma$ est la taille de la liste partant de $arg[0]$ dans l'état Σ et n_{Σ_T} est la taille de la liste partant de $current$ dans l'état Σ. Après l'éxécution de l'instruction, on sait $\rho(current) = \sigma(\rho(arg))(0)$, et donc que ${n^0}_{\Sigma_T} = n_{\Sigma_T^{'}}$ On a donc bien $n_{\Sigma_T^{'}};6 <_{lex} {n^0}_{\Sigma_T};7$

— 1 : $addrRm$:= e : Le tas n'est pas modifié, ni la variable arg. La liste de $arg[0]$ n'est pas modifiée. L'invariant se prouve à l'aide de la règle (**assign**) et de la propriété actuelle. L'équivalence est conservé car le tas n'est pas modifié et la variable $addrRm$ est une variable réservée, elle n'entre pas dans le calcul du graphe atteignable. Équivalence : : La nouvelle mesure $n_{\Sigma_T^{'}};5$. Le variable $current$ n'a pas été modifié, ni le tas donc $n_{\Sigma_T^{'}} = n_{\Sigma_T}$. On a donc bien $n_{\Sigma_T^{'}};5 <_{lex} n_{\Sigma_T};6$.

— 1.5 : $tmp := current[F\_ADD]$

— 2: while $tmp!$ = $addrRm$ do $oldCurrent := current; current := current[F\_NEXT]; tmp := current[F\_ADD]$: Le tas n'est pas modifié, ni la variable arg. La liste de $arg[0]$ n'est donc pas modifiée, ce qui permet de prouver l'équivalence et la plupart des propriétés de l'invariant. La propriété 8.6 est plus délicate. D'après le lemme 8.7, l'instruction suivante est soit $instrAt(removeList, 2.0)$ ou $instrAt(removeList, 3)$. Or, la propriété $prop_{removeList}(\Sigma, 2.0)$ est quasiment identique à $prop_{removeList}(\Sigma, 3)$, la preuve est donc quasiment identique. La propriété est quasiment la même que $prop_{removeList}(\Sigma, 2)$. D'après la règle $\mathbf{loop_t}$, le tas et l'environnement local ne sont pas modifiés. La propriété est donc quasiment conservée. Seule une propriété diffère ente 2.0 et 3.

  — 2.0 , $\sigma(\rho(current))(F\_ADD) \neq \ell$: On sait dans ce cas que la condition de la boucle est vérifié, donc que $current[F\_ADD]! = addrRm$. Or on sait que $\rho(addrRm) = \ell$. Donc la condition est vérifiée.

  — 3 , $\sigma(\rho(current))(F\_ADD) = \ell$: La preuve est le symétrique du point précédent.

  La mesure courante est $n_{\Sigma_T};5$. L'instruction suivante est soit $instrAt(removeList, 2.0)$ soit $instrAt(removeList, 3)$ suivant que la boucle soit exécutée ou non :

  — Dans le premier cas, la nouvelle mesure est $n_{\Sigma_T} - 1;7$. En effet nous sommes dans la boucle, le calcul de la mesure est différente de la position actuelle. D'après la règle sémantique $\mathbf{loop_t}$, le bloc correspondant au corps de la boucle est ajouté au bloc courant. D'après la définition de $removeList$, le bloc de la boucle comporte deux instructions. Le

nombre d'instruction restante passe donc à 7. La variable *current* n'est pas modifiée, nous avons donc $n_{\Sigma_T} = n_{\Sigma'_T}$. La mesure décroît donc bien $n_{\Sigma'_T} - 1; 7 <_{lex} n_{\Sigma_T}; 5$.

- Dans le second cas, la nouvelle mesure est $n_{\Sigma'_T}; 4$. Comme *current* n'a pas été modifié, ni le tas, d'après l'instruction, on sait que $n_{\Sigma'_T} = n_{\Sigma_T}$. Nous avons bien $n_{\Sigma'_T}; 4 <_{lex} n_{\Sigma_T}; 5$.

— 2.0 : $oldCurrent := current$ Le tas n'est pas modifié, ni la variable *arg*. L'invariant est donc une déduit du précédent et de la règle (**assign**). L'équivalence des états se prouve car le tas n'est pas modifié et la variable *oldCurrent* est une variable réservée. La mesure courante est $n_{\Sigma_T} - 1; 7$. La mesure du nouvelle état est $n_{\Sigma'_T} - 1; 6$. On sait que ni *current*, ni le tas n'est modifié, donc $n_{\Sigma'_T} = n_{\Sigma_T}$. Nous avons donc bien $n_{\Sigma'_T} - 1; 6 <_{lex} n_{\Sigma_T} - 1; 7$

— 2.1 : $current := current[F\_NEXT]$: Le tas n'est pas modifié, ni la variable *arg*. La liste de $arg[0]$ n'est donc pas modifiée, ce qui permet la preuve de l'invariant et de l'équivalence. La mesure courante est $n_{\Sigma_T} - 1; 6$. D'après le lemme 8.7, la prochaine instruction est en position 2. La nouvelle mesure est donc $n_{\Sigma'_T}; 5$. D'après l'instruction et les propriétés, on sait que la taille de la liste partant de la variable *current* décroît de 1 après l'exécution de l'instruction. Nous avons donc $n_{\Sigma_T} - 1 = n_{\Sigma'_T}$. Nous avons donc bien $n_{\Sigma'_T}; 6 <_{lex} n_{\Sigma_T} - 1; 5$.

— 2.2 : $tmp := arg[0]$

— 3 : `if` $arg[0] = current$ `then` $arg[0] := current[F\_NEXT]$ `else` $oldCurrent[F\_NEXT] := current[F\_NEXT]$ Le tas et l'environnement ne sont pas modifiés, l'invariant et l'équivalence restent donc valides. La mesure courante est $n_{\Sigma_T}; 4$. D'après le lemme 8.7, la prochaine instruction est en position 3.0.0 ou 3.1.0. Dans ces deux cas la mesure est identique. La variable *current* et la tas ne sont pas modifiés, nous avons donc $n_{\Sigma_T} = n_{\Sigma'_T}$. La nouvelle mesure est $n_{\Sigma'_T}; 2$, et nous avons donc bien $n_{\Sigma'_T}; 2 <_{lex} n_{\Sigma_T}; 4$.

— 3.0.0 : $tmp := current[F\_NEXT]$: Le contenu de *tmp* devient égale à celui de $current[F\_NEXT]$.

— 3.0.1: $arg[0] := tmp$ D'après la propriété associée, il existe une liste non vide partant de *current*. Par définition de la liste, il existe une liste partant de $\sigma(\rho(current))(F\_NEXT)$, donc c'est bien une liste que reçoit $arg[0]$ par l'intermédiaire de *tmp* . La propriété $listdp_{\sigma,\rho} \cup lista_{\sigma,\rho} \cup \rho(arg) \cup \rho(current) = dom(\sigma)$ est valide car la liste est modifiée, le noeud dont l'adresse est contenue dans *current* est supprimé de la liste. Or on sait d'après la propriété que $\sigma(\rho(current))(F\_ADD) = \ell$, on en déduit donc la propriété. La propriété $listdp_{\sigma,\rho} \cap lista_{\sigma,\rho} \cap \rho(arg) \cap \rho(current) = \emptyset$ est valide car la variable *current* contenait un élément de la liste. Il était présent une seule fois dans la liste. Cet élément est supprimé Un élément est supprimé de la liste. L'intersection vide est conservée de l'invariant 8.5. L'équivalence de tas $\Sigma'_T \sim^{heap} \Sigma_S$ est

valide car de la me manière que pour l'invariant 8.6, on sait que la liste d'exclusion est identique entre Σ et $\Sigma_T^{'}$, donc l'invariant est conservé. La mesure courante est $n_{\Sigma_T};2$. La variable *current* et la tas ne sont pas modifiés, nous avons donc $n_{\Sigma_T} = n_{\Sigma_T'}$. La nouvelle mesure est $n_{\Sigma_T'};1$, et nous avons donc bien $n_{\Sigma_T'};1 <_{lex} n_{\Sigma_T};2$.

— 3.1.0 : $tmp := current[F\_NEXT]$
— 3.1.1 : $oldCurrent[F\_NEXT] := tmp$:On sait d'après la propriété que $current[F\_NEXT]$ contient une liste, cette liste est un suffixe de la liste de $arg[0]$. On sait aussi d'après la propriété que $\sigma(\rho(oldCurrent))(F\_NEXT) = \rho(current)$, donc *oldCurrent* est le noeud précédent. L'instruction supprime donc le noeud *current* de la liste. Il existe donc toujours une liste dans $arg[0]$. La propriété $listdp_{\sigma,\rho} \cup lista_{\sigma,\rho} \cup \rho(arg) \cup \rho(current) = dom(\sigma)$ est valide car la liste est modifiée, le noeud dont l'adresse est contenue dans *current* est supprimé de la liste. Or on sait d'après la propriété que $\sigma(\rho(current))(F\_ADD) = \ell$, on en déduit donc la propriété. La propriété $listdp_{\sigma,\rho} \cap lista_{\sigma,\rho} \cap \rho(arg) \cap \rho(current) = \emptyset$ est vérifiée car un élément est supprimé de la liste. L'intersection vide est conservée. L'équivalence de tas $\Sigma_T^{'} \sim^{heap} \Sigma_S$ est conservé car de manière similaire à la preuve de l'invariant 8.6 , on sait que la liste d'exclusion est identique entre Σ et $\Sigma_T^{'}$. L'invariant est donc conservé. La mesure courante est $n_{\Sigma_T};2$. La variable *current* et la tas ne sont pas modifiés, nous avons donc $n_{\Sigma_T} = n_{\Sigma_T'}$. La nouvelle mesure est $n_{\Sigma_T'};1$, et nous avons donc bien $n_{\Sigma_T'};1 <_{lex} n_{\Sigma_T};2$.
— 4 : $dispose(current)$: On sait qu'un seul élément de la liste pointe vers $\rho_T(current)$. On sait que *oldCurrent* pointe vers cette élément. Le reste de la liste n'est donc pas modifié par cette suppression, l'invariant est conservé. On sait d'après la propriété que $listdp_{\sigma,\rho} \cap lista_{\sigma,\rho} \cap \rho(arg) \cap \rho(current) = \emptyset$. On sait donc que *current* n'est pas dans la liste. Donc la suppression de l'adresse contenue dans cette variable établit la propriété 8.4. L'équivalence de tas$\Sigma_T^{'} \sim^{heap} \Sigma_S$ est valide car on sait que $reachableGraph_{\sigma_T}(\rho_T, listp_{\sigma,\rho} \cup \rho_T(current)) = reachableGraph_{\sigma_S}(\rho_S, \emptyset)$, il faut prouver que $reachableGraph_{\sigma_T}(\rho_T, listp_{\sigma,\rho}) = reachableGraph_{\sigma_S}(\rho_S, \emptyset)$.

□

Vers la correction de la compilation de AFJ vers LUFJ

9

Sommaire

Dans ce chapitre, nous nous intéressons à la correction de la compilation d'un programme *AFJ* en un programme *LUFJ*. Les définitions et la structure de la preuve sont inspirées de la compilation d'un langage séquentiel exposée dans le chapitre précédent, notamment la représentation abstraite du tas, et le traitement des blocs d'instructions instrumentées. Néanmoins dans cette nouvelle compilation, les langages sources et cibles ne sont pas identiques : le langage *AFJ* définit des sections atomiques de manière syntaxique, instructions absentes du langage *LUFJ*. Nous nous intéressons donc à la préservation de ce comportement par la compilation, qui transforme ces sections atomiques en code faisant appel à des verrous. De plus le parallélisme existe dans ces deux langages. Cela a donc un impact dans l'étude d'un état d'un programme, avec la vérification des instructions actives. Nous nous intéressons tout d'abord aux différentes définitions nécessaires à la preuve de conservation de la sémantique, avec entre autre la caractérisation des structures de données support à la compilation, introduites dans le chapitre 7. Nous définissons ensuite l'équivalence entre états *AFJ* et *LUFJ*. Enfin nous étudions la preuve de préservation des comportements d'un programme.

9.1 Prédicats sur les états

Pour un langage $\mathcal{L} \in \{\mathrm{L}, \mathrm{A}\}$ où L est utilisé pour désigner LUFJ, A pour désigner AFJ, nous utilisons les notations suivantes:

— $\Sigma^{\mathcal{L}}$est un état de la sémantique opérationnelle du langage $\mathcal{L}$ telle que définies respectivement aux chapitres 5 et 6, avec possible indice,

— si p est un programme du langage $\mathcal{L}$, $\Sigma_1^{\mathcal{L}}$ et $\Sigma_2^{\mathcal{L}}$ deux états et e un évènement de la sémantique opérationnelle du langage, alors on note $\vdash_p \Sigma_1^{\mathcal{L}} \xrightarrow{e}_{\mathcal{L}} \Sigma_2^{\mathcal{L}}$ une étape de réduction de la sémantique opérationnelle du langage $\mathcal{L}$ produisant l'évènement e.

Nous avons vu dans le chapitre 7 que chaque processus léger est associé à une structure de données *info* le représentant (cf. figure 7.2). Une structure *info* contient deux adresses vers ses voisins *info*, un verrou, et une adresse vers sa bulle. Ils sont utilisés en tant qu'éléments d'une liste doublement chaînée. Une telle liste représente les processus légers s'exécutant dans une section atomique. La définition formelle du prédicat vérifie que les différents éléments de la structure *info* sont correctement définis. Comme c'est une liste cyclique doublement chaînée, nous ne pouvons pas réutiliser la définition 8.12, nous devons donc introduire une définition d'une telle liste.

Notre liste est paramétrée par le type de la donnée contenue dans chacun des maillons (champ *DATA*). Une fonction *isValid* détermine si la donnée est correcte. Deux définitions sont nécessaires : *cyclicDoubleLinked* est la définition de notre liste doublement chaînée, qui se base sur la définition auxiliaire *cyclicDoubleTail*. Le prédicat *cyclicDoubleLinked* permet de faire le cycle sur une liste doublement chaînée. Les figures 9.1 et 9.2.

Une liste cyclique doublement chaînée est définie ainsi :

$$\frac{\begin{array}{c}\sigma(fop)(PREV) = prev \\ \sigma(fop)(NEXT) = root \\ \sigma(fop')(DATA) = d \\ isValid_{\sigma,\Lambda}(d)\end{array}}{cyclicDoubleTail_{\sigma,\Lambda}(root, fop, [(prev, d, root)])}$$

$$\frac{\begin{array}{c}cyclicDoubleTail_{\sigma,\Lambda}(root, fop, xs) \\ prev \notin (map\ first\ xs) \\ \exists xs = (fop', \_, \_) :: xs' \\ \sigma(fop')(PREV) = prev \\ \sigma(fop')(NEXT) = fop \\ \sigma(fop')(DATA) = d \\ isValid_{\sigma,\Lambda}(d)\end{array}}{cyclicDoubleTail_{\sigma,\Lambda}(root, fop', (prev, d, fop) :: xs))}$$

$$\frac{}{cyclicDoubleLinked_{\sigma,\Lambda}(null, [])}$$

$$\frac{\begin{array}{c}\exists a, xs', xs = ((a, \_, \_) :: (xs' + +(\_, \_, a) :: [(\_, \_, \_)]) \wedge \\ cyclicDoubleTail_{\sigma,\Lambda}(root, root, xs)\end{array}}{cyclicDoubleLinked_{\sigma,\Lambda}(root, xs)}$$

Grâce à cette définition nous pouvons désormais définir ce qu'est une liste d'éléments *info*. Pour cela, il est nécessaire de définir la fonction *isValid*.

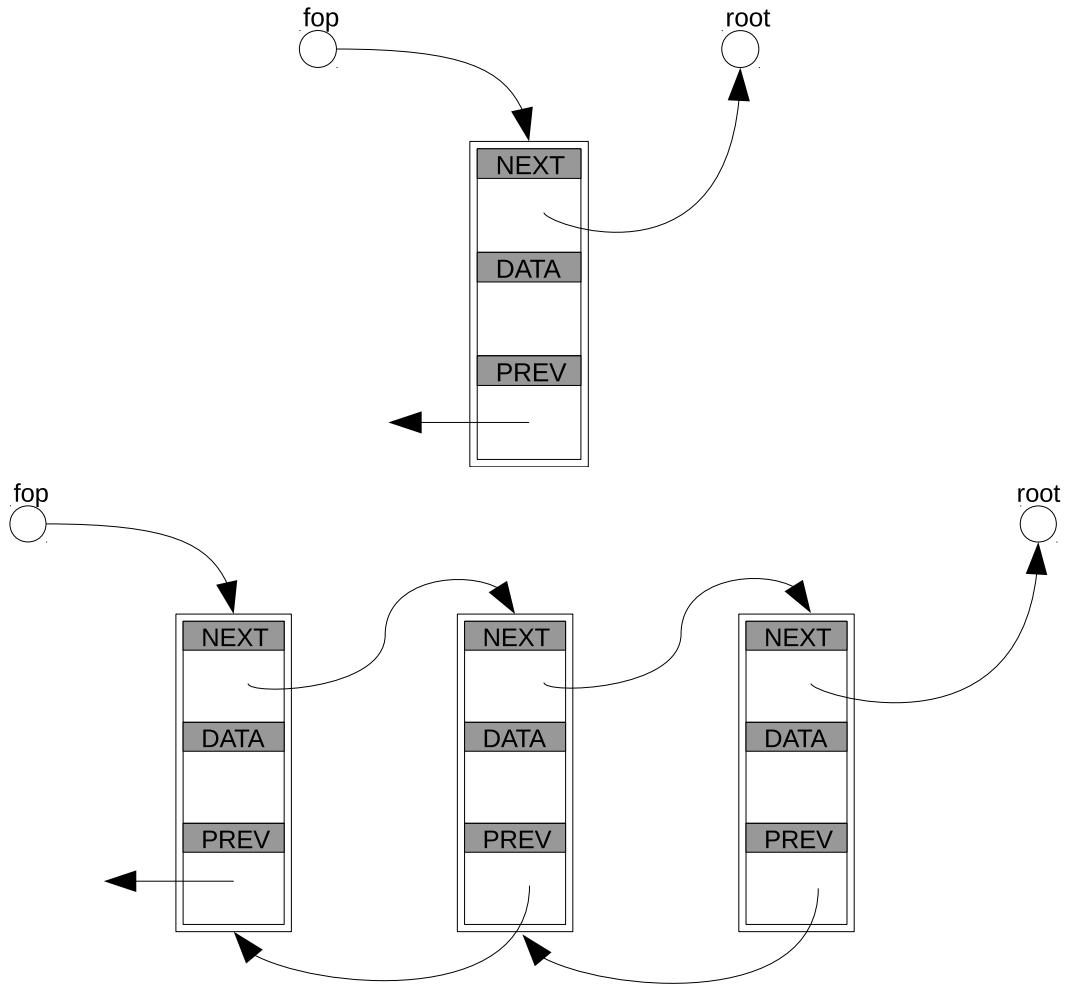


FIGURE 9.1 – *cyclicDoubleTail*

Définition 9.1

$$\frac{\exists xs_b,\ listeBulle_{\sigma,\Lambda}(\ell_2, xs_b) \wedge \ell_3 \in dom(\Lambda)}{isValid_{\sigma,\Lambda}(data)}$$

$$\frac{cyclicDoubleLinked_{\sigma,\Lambda}(root, xs)}{listeInfo_{\sigma,\Lambda}(root, xs)}$$

Le prédicat *isInfo* détermine si une adresse fait référence à une liste d'*infos*.

Définition 9.2

$$\frac{\ell \neq \texttt{null} \ \wedge\ \exists xs,\ listeInfo_{\sigma,\Lambda}(\ell, xs)}{isInfo_{\sigma,\Lambda}(\ell)}$$

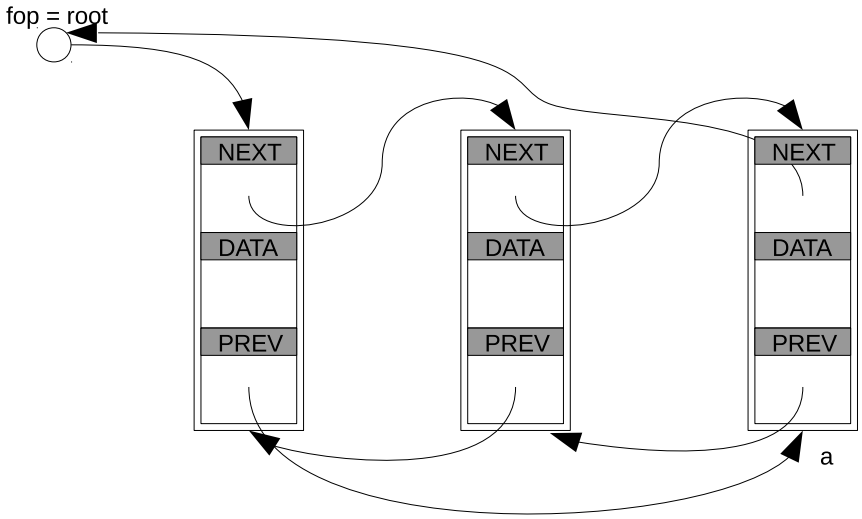


FIGURE 9.2 – *cyclicDoubleLinked*

Également introduite dans le chapitre 7, nous savons que chaque section atomique est représentée sous forme d'une structure mémoire appelée bulle (cf figure 7.1). Une section atomique peut avoir une sous-section et une section parente. Une bulle possède donc un pointeur vers sa section parente et sa sous section. Une bulle est donc un élément d'une liste doublement chaînée. Elle contient également une adresse d'une liste d'*infos* et deux verrous. La définition suivante formalise le prédicat caractérisant la présence d'une telle liste dans la mémoire.

Définition 9.3 *Un liste de structure de bulles est définie de la façon suivante :*

$$\frac{cyclicDoubleLinked_{\sigma,\Lambda}(root, xs)}{listeBulle_{\sigma,\Lambda}(root, xs)}$$

Nous définissons des fonctions pour obtenir la liste des bulles à partir d'un environnement local, d'un processus léger et du tas. Chaque processus léger possède dans son environnement local une variable pointant vers sa structure *info*. Cette structure *info* possède un pointeur vers sa section atomique, sa bulle. Grâce au double chaînage de la liste de bulles, il est possible de trouver la tête de la liste, à partir de tout élément de celle-ci. La fonction *headOfList*() retourne la tête d'une liste à partir d'une adresse de bulle, en remontant dans la liste. La fonction *getlistBubble*() retourne la liste de bulles à partir d'un environnement local et du tas, grâce à la fonction précédente. Cette fonction est étendue avec *getlistBubbleState*() qui prend cette fois en paramètre un état d'un programme LUFJ.

$$\begin{array}{rcl} headOfList_{\sigma}(\ell) & = & match\ \sigma(\ell)[F\_PREVIOUS]\ with \\ & & \mid null \Rightarrow \ell \\ & & \mid \ell_a \Rightarrow headOfList_{\sigma}(\ell_a) \end{array}$$

$$getlistBubble_{\sigma}(\rho) = headOfList_{\sigma}(\sigma(\rho(info))[F\_BUBBLE]))$$

La fonction *listPointerListInfo* retourne pour une liste d'éléments *info* les adresses de chacun des éléments.

La fonction *listeExclusion*() retourne l'ensemble des adresses des bulles et de toutes les structures *info* à partir d'un environnement local et du tas. Ainsi on obtient les adresses des structures allouées uniquement pour le support à l'exécution.

$$listPointerListInfo_{\sigma,\Lambda}(\ell) = \exists xs,\ listeInfo_{\ell,\Lambda}(\sigma, xs) \wedge \ell :: (map\ fourth\ xs)$$

$$listeExclusion_{\sigma}(\rho) = \left\{ \rho(arg) \cup xs \cup \bigcup xs_i \mid \begin{array}{l} \exists \ell,\ getlistBubble_{\sigma}(\rho) = \ell\ \wedge\ listeBulle_{\sigma,\Lambda}(\ell, xs) \wedge \\ (\forall \ell_i \in map\ fourth\ xs\ \wedge\ listeInfo_{\sigma,\Lambda}(\ell_i, xs_i)) \end{array} \right\}$$

Définition 9.4 *Il est utile pour un processus léger donné de connaître son instruction active. La fonction $instrOfThread_{\phi}t$ à partir d'un ensemble de processus légers ϕ et d'un identifiant de processus léger t, retourne l'instruction en tête du bloc d'instructions de ce processus léger t.*

$$instrOfThread_{\phi}(b, \rho)^{tid} = hd(b)$$

Définition 9.5 *Le prédicat $isInBlock_{\phi}(b, pos)$ est un prédicat déterminant s'il existe un processus léger dans l'ensemble ϕ dont l'instruction en tête est celle du bloc d'instruction b en position pos.*

$$\frac{\exists t \in \phi,\ instrOfThread_{\phi}t = b_{pos}}{isInBlock_{\phi}(b, pos)}$$

On étend cette notation, aux états (à la place d'un ensemble de processus légers).

Définition 9.6 *Le prédicat suivant indique le lien entre un processus léger et une structure info. Il vérifie que l'adresse donnée est bien celle associée dans l'environnement local du processus léger à la variable info, et que cette adresse contient bien une structure info.*

$$threadInfo_{\sigma,\Lambda}(t, \ell) = \rho(info) = (\ell\ \wedge\ isInfo_{\sigma,\Lambda}(\ell)\ où\ \rho = snd(t))$$

Nous avons dans état d'un programme AFJ, l'identifiant du processus léger responsable de l'ouverture d'une section atomique, appelé processus léger propriétaire de la section. Cette information n'est pas directement accessible dans un état d'un programme LUFJ. Néanmoins il est possible de calculer en inspectant le bloc d'instructions restantes d'un processus léger, en s'intéressant aux instructions de déverrouillage non associées à une de verrouillage.

```
getThreadFromInfo(info : info)(phi : threads) : (nat, thread) :=
  find phi info

mapOfThreads(listInfo : list info)(phi : threads) : threads :=
  let listThreads := list.map(getThreadFromInfo phi) listInfo in
    list.fold_left(fun map (tid, t) → map.add tid t) emptyMap listThreads

threadsOfBubble (B : Bubble)(phi : threads)(i : nat) : threads :=
  let bubble := getBubble B i in
   let listInfo := B.first in
    mapThreads listInfo phi

threadOpenSection (B : Bubble)(phi : threads)(i : nat) : option thread :=
  let phi' := threadsOfBubble B phi i in
    prefixNonAtomic phi'
```

FIGURE 9.3 – *Fonctions liées à la recherche de processus légers ayant des sections ouvertes*

Définition 9.7 *La fonction $threadOwner_\sigma(n, \phi)$ retourne le processus léger présent dans ϕ propriétaire de la section numéro n. Pour trouver ce propriétaire, la fonction inspecte le nombre d'ouverture et de fermeture de section de chaque processus léger.*

La fonction *depth* retourne nombre de niveau d'une bulle.

La fonction *threadOpenSection* présentée dans la figure 9.3 avec une syntaxe à la Coq, retourne le processus léger s'il existe présent dans la bulle de niveau i, qui a ouvert une section atomique ouverte, c'est à dire qui contient dans ses instructions un `unlock` sans `lock` associé.

La fonction *threadsOfBubble* à partir d'un ensemble de bulles, d'un ensemble de processus légers et d'un entier, retourne le sous ensemble de processus légers qui sont présent dans la bulle de niveau donné. La fonction *getBubble* retourne la bulle de niveau demandé. La fonction *mapOfThreads* prend en paramètre une liste de structures *info* et un ensemble de processus légers. Il retourne le sous ensemble de celui-ci qui possède dans son environnement local une variable *info* contenant un élément de la liste de structures *info* donnée en paramètre. La recherche de ces processus est effectuée par la fonction *getThreadFromInfo*.

La fonction *prefixNonAtomic* recherche le processus léger parmi un ensemble qui contient une instruction `unlock` sans `lock` associé. Elle n'est pas présentée dans la figure 9.3.

Les fonctions présentées dans la figure 9.4 permettent de trouver les processus légers propriétaires de sections atomiques. La fonction *owners* retourne la liste des processus légers propriétaires, où la position de chaque élément dans la liste correspond au niveau dans la bulle. Elle se base sur la fonction *ownersaux* qui retourne les

```
Fixpoint ownersaux (B : Bubble)(i : nat)(phi : threads)(l : list thread){struct i} : list thread :=
  let tid := threadOpenSection B phi i in
    match i, tid with
    | 0, Some t ⇒ t :: l
    | 0, None ⇒ match l with
                | nil ⇒ l
                | a :: l' ⇒ a :: l
                end
    | S n, Some t ⇒ let l' := t :: l in (ownersaux B n phi l') ++l'
    | S n, None ⇒ (ownersaux B n phi l) ++l
end.

Definition owners (B : Bubble)(phi : threads) : list thread :=
  let depth := depth B in
    ownersaux B depth phi nil.

Definition owner (B : Bubble)(i : nat)(phi : threads) : option thread :=
  nth_error (owners B phi) i.

Definition threadOwner(n : nat)(phi : threads)(h : heap) : option thread :=
  owner(getListBubble(h, snd(phi(t_ε))), n)
    nth_error (owners B) i.
```

FIGURE 9.4 – *Fonctions pour déterminer les processus légers propriétaires de sections atomiques*

processus légers propriétaires des i premiers niveaux dans la bulle. Pour chaque niveau, nous regardons si un processus léger a une section ouverte grâce à la fonction *threadOpenSection*. S'il existe un tel processus léger, il est propriétaire de ce niveau. Sinon, le propriétaire est le même que la sous-section immédiate.

La fonction *threadOwner* prend en paramètre des éléments d'un état LUFJ, c'est donc cette fonction qui est utilisée. La fonction *owner* effectue le calcul de la recherche du processus léger propriétaire, mais prend une bulle en paramètre, élément qui n'est pas un élément d'un état LUFJ. La fonction *threadOwner* consiste donc à obtenir la structure de bulle à partir du tas et de l'ensemble des processus légers, pour pouvoir appeler *owner*.

Définition 9.8 *La fonction blockClose(t, n) retourne le bloc d'instructions du processus léger t après le n^{eme} bloc close le plus à droite qui ne possède pas de bloc open associé.*

Définition 9.9 *La fonction prefixPendingClose(b) retourne le bloc d'instructions issu de b à gauche du premier bloc d'instructions close pendant. Un bloc d'instructions close pendant est un bloc close sans bloc open associé.*

Définition 9.10 *La fonction removePrefixSuffix(b) supprime de b le préfixe s'il est égal à un suffixe stricte d'un bloc d'instructions instrumentées.*

9.2 Correction de la passe de compilation

9.2.1 Équivalence d'états

Nous pouvons définir l'équivalence entre un état entre un état *LUFJ* Σ_T et un état *AFJ* Σ_S. Nous décomposons cette équivalence entre les différents composants d'un état. Nous étendons les états AFJ en leur ajoutant $\mathcal{A}$. Les deux états contiennent donc un élément $\mathcal{A}$, mais il n'intervient pas dans les prédicats. Par volonté de simplifier la notation, nous ne le faisons pas apparaître systématiquement.

Équivalence de tas L'équivalence de tas se base sur la représentation abstraite du tas calculé par la fonction *reachableGraph*. Il y a une représentation du graphe par processus léger, symbolisant les valeurs accessibles depuis l'environnement local de celui-ci. Chacun de ces graphes accessibles doit être identiques à celui calculé pour le processus léger correspondant de l'état source. Le calcul du graphe prend en paramètre une liste d'adresses à ne pas prendre en compte (variables nécessaires à l'instrumentation, non présentes dans l'état source). Cette liste $listeExclusion_{\sigma_T}(\rho_T)$ regroupe la liste des bulles et des *info*s. Cette liste peut être légèrement modifiée en fonction de l'instruction active d'un processus léger. Par exemple lorsqu'une nouvelle section est ouverte, la nouvelle bulle n'est pas immédiatement ajoutée à la liste des bulles, il faut donc la prendre en compte pendant l'intervalle entre le moment où elle est créée en mémoire et le moment où elle est ajoutée à la liste des bulles. Dans la propriété suivante, les trois premiers cas correspondent à ces cas particuliers, et le dernier au cas général.

On définit tout d'abord les différentes gardes qui permettent d'exprimer ces différentes positions des processus légers dans le code. La condition $threadClosing((\phi, \sigma_T, \Lambda), t, B)$ permet d'exprimer que le processus léger t est dans un bloc de fermeture, dans la bulle B. Un état $\Sigma = (\sigma, \phi, \Lambda, \mathcal{A})$ et $t = (b, \rho)^{tid}$.

Nous utilisons la notation $\cdot$ lorsque nous ne voulons pas nommer un élément dont ne nous servons pas. Par exemple $t = (\cdot, \rho)$ est équivalent à $\exists b, t = (b, \rho)$.

$$
\begin{aligned}
\mathit{Initialised}(\Sigma, t) &= \exists\ \mathit{pos},\ 0 < \mathit{pos} < 4,\ \wedge\ \mathit{isInBlock}_{\Sigma}(t, \mathit{initialize}, \mathit{pos}) \\
\mathit{Open}(\Sigma, t) &= \exists\ \mathit{pos},\ 1 < \mathit{pos} < 4,\ \wedge\ \mathit{isInBlock}_{\Sigma}(t, \mathit{open}, \mathit{pos}) \\
\mathit{Close}(\Sigma, t) &= \exists\ \mathit{pos},\ 2 \leq \mathit{pos} < |\mathit{close}|,\ \wedge\ \mathit{isInBlock}_{\Sigma}(t, \mathit{close}, \mathit{pos}) \\
\mathit{ForkPB}(\Sigma, t) &= \exists\ \mathit{pos},\ 4 \leq \mathit{pos} < 5,\ \wedge\ \exists e,\ \mathit{isInBlock}_{\Sigma}(t, \mathit{forkPB}(e), \mathit{pos}) \\
\mathit{ForkCB}(\Sigma, t) &= \exists\ \mathit{pos},\ 0 \leq \mathit{pos} < 2,\ \wedge\ \exists e,\ \mathit{isInBlock}_{\Sigma}(t, \mathit{forkCB}(e), \mathit{pos}) \\
\mathit{general}(\Sigma, t) &= \neg(\mathit{Initialised}(\Sigma, t)\ \vee\ \mathit{Open}(\Sigma, t)\ \vee\ \mathit{Close}(\Sigma, t)\ \vee \\
&\quad \mathit{ForkPB}(\Sigma, t)\ \vee\ \mathit{ForkCB}(\Sigma, t)) \\
\mathit{RemoveInfo1}((\phi, \sigma_T, \Lambda), t) &= \exists\ \mathit{pos},\ 5 < \mathit{pos} < |\mathit{remove}|\ \wedge\ \mathit{isInBlock}_{\Sigma}(t, \mathit{remove}, \mathit{pos}) \\
\mathit{RemoveInfo}((\phi, \sigma_T, \Lambda), t, xs) &= \mathit{RemoveInfo1}((\phi, \sigma_T, \Lambda), t)\ \vee \\
&\quad ((\mathit{isInBlock}_{\Sigma}(t, \mathit{open}, 5))\ \wedge\ (\forall \rho_T,\ \rho_T = \mathit{snd}(t)\ \wedge \\
&\quad \sigma_T(\rho_T(\mathit{info}))[\mathit{F\_BUBBLE}] = \ell\ \wedge\ \mathit{listeBulle}_{\sigma_T, \Lambda}(\ell, xs))) \\
\mathit{bubbleCreated}(\Sigma, t) &= \exists\ \mathit{pos},\ 3 < \mathit{pos} < |\mathit{open}|,\ \wedge\ \mathit{isInBlock}_{\Sigma}(t, \mathit{open}, \mathit{pos}) \\
\mathit{threadClosing}(\Sigma, t, \ell_B) &= \mathit{Close}(\Sigma, t)\ \wedge\ \forall \rho, t = (\cdot, \rho)\ \wedge \\
&\quad \mathit{listeBulle}_{\sigma_T, \Lambda}(\ell, \ell_B)\ \wedge\ \rho(\mathit{info}) \in \sigma(\ell_B)[\mathit{first}]
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{generalStruct}((\phi_T, \sigma_T, \Lambda), t) \equiv (\ \nexists t, \mathit{RemoveInfo1}((\phi_T, \sigma_T, \Lambda), t)\ \wedge \\
&\nexists t,\ \nexists xs',\ \mathit{RemoveInfo}((\phi_T, \sigma_T, \Lambda), t, xs')\ \wedge \\
&\nexists t,\ \nexists xs',\ \mathit{initialize}(\phi_T, \sigma_T, \Lambda) t xs')
\end{aligned}
$$

Nous pouvons maintenant définir l'équivalence de tas :

$$\forall t_T, t_S, \rho_T, \rho_S,\ t_T \in \phi_T,\ \wedge\ t_S \in \phi_S\ \wedge\ t_T, \sigma_T \equiv^{thread} t_S\ \wedge\ \rho_T = snd(t_T)\ \wedge\ \rho_S = snd(t_S)$$
$$\Rightarrow$$
$$(Open(\Sigma, t_T) \Rightarrow$$
$$reachableGraph_{\sigma_T}(\rho_T, listeExclusion_{\sigma_T}(\rho_T) \cup \rho(nBubble)) = reachableGraph_{\sigma_S}(\rho_S, \emptyset)) \wedge$$

$$(Close(\Sigma, t_T) \Rightarrow$$
$$reachableGraph_{\sigma_T}(\rho_T, listeExclusion_{\sigma_T}(\rho_T) \cup \rho(cell)) = reachableGraph_{\sigma_S}(\rho_S, \emptyset)) \wedge$$

$$(ForkPB(\Sigma, t_T) \Rightarrow$$
$$reachableGraph_{\sigma_T}(\rho_T, listeExclusion_{\sigma_T}(\rho_T) \cup \rho(childInfo)) = reachableGraph_{\sigma_S}(\rho_S, \emptyset)) \wedge$$

$$(ForkCB(\Sigma, t_T) \Rightarrow$$
$$reachableGraph_{\sigma_T}(\sigma(\rho_T(arg))(1), listeExclusion_{\sigma_T}(\rho_T) \cup \rho(childInfo)) = reachableGraph_{\sigma_S}(\rho_S, \emptyset)) \wedge$$

$$(RemoveInfo1(\Sigma, t_T) \Rightarrow$$
$$reachableGraph_{\sigma_T}(\rho_T, listeExclusion_{\sigma_T}(\rho_T) \cup \rho(info)) = reachableGraph_{\sigma_S}(\rho_S, \emptyset)) \wedge$$

$$general(\Sigma, t_T) \Rightarrow$$
$$reachableGraph_{\sigma_T}(\rho_T, listeExclusion_{\sigma_T}(\rho_T)) = reachableGraph_{\sigma_S}(\rho_S, \emptyset))$$

$$\sigma_T, \phi_T, \Lambda \equiv^{heap} \sigma_S, \phi_S$$

Équivalence d'environnements locaux L'équivalence d'environnements locaux se décompose en deux cas, suivant que l'instruction active du processus de l'état cible soit à la première instruction du bloc *forkCB* ou non. Le cas général, lorsque cette condition n'est pas vérifiée, consiste à vérifier l'égalité entre les variables non réservées des deux environnements qui ne sont ni des adresses, ni des identifiants de processus léger. La fonction *filterVar*() sur un environnement ne conserve que les variables contenant une valeur différente d'une adresse. Les variables dont le nom est réservé sont également exclues. Lorsque l'instruction active du processus de l'état cible est égale à la première du bloc *forkCB*, l'environnement local ne contient que l'argument de la méthode. Cet argument, *arg*, fait référence à un enregistrement dans le tas, contenant sa structure *info*, et l'argument original de la méthode. Il faut donc comparer cette valeur, uniquement si c'est une valeur différente d'une adresse et d'un identifiant de processus léger, avec celle dans l'environnement local source.

$$\frac{(hd(b_T) = forkCB_0\ \wedge\ \exists v \in \mathbb{N}\ \sigma(\rho_T(arg))[1] = v\ \Rightarrow \rho_S(x) = v) \vee (hd(b_T) \neq forkCB_0 \Rightarrow filterVar(\rho_T) = filterVar(\rho_S))}{b_T, \rho_T, \sigma \equiv^{env} \rho_S}$$

Équivalence de blocs d'instructions L'équivalence de blocs d'instructions nécessite un traitement particulier pour chacun des blocs cibles et sources. Dans l'état source, un processus léger propriétaire d'une section ne contient pas l'intégralité de ses instructions. La partie exécutée après la fin de la section est sauvegardée dans le contexte de la bulle. Cette séparation n'est pas présente dans les processus légers de l'état cible. Il est donc nécessaire de considérer uniquement le bloc d'instructions présent juste avant le premier bloc d'instructions instrumentées *close* pendant. Un bloc *close* est pendant s'il n'a pas de bloc *open* associé. Le processus léger cible peut être dans un bloc d'instructions instrumentées entamé. Il est nécessaire de ne pas prendre en compte ces instructions. Ces deux modifications sont faites par les fonctions *removePrefixSuffix*() et *prefixPendingClose*(). La fonction *removePrefixSuffix*() supprime le premier suffixe de bloc d'instruction instrumenté. La fonction *prefixPendingClose*() retourne le bloc d'instruction avant le premier bloc *close* pendant. La comparaison consiste donc à vérifier l'égalité entre le bloc cible modifiée et le bloc source compilé.

$$\frac{\mathit{removePrefixSuffix}(\mathit{prefixPendingClose}(b_T)) = \mathit{removePrefixSuffix}(\mathcal{C}(b_S))}{b_T \equiv^{block} b_S}$$

Équivalence de processus légers L'équivalence de processus légers consiste à vérifier l'équivalence des blocs d'instructions et des environnements locaux.

$$\frac{\begin{array}{c} t_T = (b_T, \rho_T)^{tid} \wedge t_S = (b_S, \rho_S)^{tid} \wedge \\ b_T \equiv^{block} b_S \wedge \\ b_T, \rho_T, \sigma \equiv^{env} \rho_S \end{array}}{t_T, \sigma \equiv^{thread} t_S}$$

Équivalence d'ensembles de processus légers L'équivalence d'un ensemble de processus se fait entre une liste de processus légers (leur représentation sous forme de la structure *info*) issue du programme cible et un ensemble de processus légers du programme source.

$$\frac{\exists \ell_{xs}, \mathit{listeInfo}_{\sigma,\Lambda}(\ell_{xs}, xs) \ \wedge \ \forall \ell_i \in xs, \exists t \in \phi_T, \ \mathit{threadInfo}_{\sigma,\Lambda}(t, \ell_i) \ \wedge \ \exists t' \in \phi_S, \ t, \sigma \equiv^{thread} t'}{xs, \phi_T, \sigma, \Lambda \ \equiv^{threads} \ \phi_S}$$

Équivalence de bulles L'équivalence de bulles se décompose en plusieurs parties

- Le cas général est utilisé lorsque la bulle de AFJ n'est pas vide. Cette équivalence consiste à comparer
  - la liste processus léger des deux bulles, dans le cas de la bulle LUFJ cela consiste à considérer la liste des structures *info*,
  - le processus léger propriétaire de ce niveau,

  - le contexte,
  - la bulle de niveau inférieur.

  La comparaison des processus légers a deux cas particuliers, si un des processus légers est en train d'ouvrir ou de fermer une section :
  - Si un processus léger est en train d'ouvrir une section, il faut le considérer comme faisant toujours partie de la bulle. Cela s'explique par le fait que nous considérons l'équivalence entre un état LUFJ dont un des processus léger est en train d'ouvrir une nouvelle bulle, et l'état AFJ *avant* l'instruction `open`.
  - Si un processus léger est en train de se fermer dans la bulle suivante, il faut considérer tous les processus légers de la bulle suivante comme faisant partie de la bulle courante. Cela est dû au fait que nous considérons l'équivalence entre un état LUFJ dont un des processus légers est en train de fermer une bulle, et l'état AFJ *apres* l'instruction `close`.
- La liste de sous-bulles est de taille un. Elle est équivalente a une bulle vide de AFJ, s'il existe un processus léger qui est en train de créer une nouvelle bulle (condition *bubbleCreated*). Cela s'explique par le fait que nous considérons l'équivalence entre un état LUFJ dont un des processus léger est en train d'ouvrir une nouvelle bulle, et l'état AFJ *avant* l'instruction `open`. Dans l'état cible la bulle existe, mais n'a pas encore d'équivalent dans l'état source.
- La liste de sous-bulles est de taille un. Elle est équivalente a une bulle vide de AFJ, s'il existe un processus léger qui est en train de fermer une nouvelle bulle (condition *Close*). Cela s'explique par le fait que nous considérons l'équivalence entre un état LUFJ dont un des processus léger est en train de fermer une bulle, et l'état AFJ *apres* l'instruction `close`. Donc dans l'état cible, la bulle existe encore, mais dans l'état source, la section n'existe plus.
- Une liste de sous-bulle de taille zéro est équivalente à une bulle vide.

$$\frac{\begin{array}{c} xs = \{prev = \ell_1, lockB = \ell_2, next = \ell_3, first = \ell_4, lockL = \ell_5\} :: xs' \wedge \\ listeBulle_{\sigma_T,\Lambda}(\cdot, xs) \wedge \\ \exists xs_l,\ listeInfo_{\sigma_T,\Lambda}(first, xs_l) \wedge \\ \\ (\exists t, RemoveInfo((\phi_T, \sigma_T, \Lambda), t, xs) \Rightarrow \\ listPointerListInfo_{\sigma,\Lambda}(first) \cup snd(t)(info), \phi_T, \sigma \ \equiv^{threads}\ \phi_S) \wedge \\ \\ (xs' = \{prev' = \cdot, lockB' = \cdot, next' = \cdot, first' = \ell'_4, next' = \cdot\} :: xs'' \\ \exists t, threadClosing((\phi_T, \sigma_T, \Lambda), t, xs') \Rightarrow \\ listPointerListInfo_{\sigma,\Lambda}(first) \cup listPointerListInfo_{\sigma,\Lambda}(first'), \phi_T, \sigma \ \equiv^{threads}\ \phi_S) \wedge \\ \\ (generalStruct((\phi_T, \sigma_T, \Lambda), t) \Rightarrow \\ listPointerListInfo_{\sigma,\Lambda}(first), \phi_T, \sigma \ \equiv^{threads}\ \phi_S) \wedge \\ \\ threadOwner_{\sigma_T}(n, \phi_T) = t_o \wedge \\ \\ t_o, \sigma_T \equiv^{thread} t \wedge \\ \\ blockClose(t_0, n - |xs'| + 2) \equiv^{block} C \wedge \\ \\ xs', \phi_T, \sigma_T, \Lambda, S\ n \ \equiv^{bubble}\ \mathrm{B}^\circ \end{array}}{(xs, \phi_T, \sigma_T, \Lambda, n) \ \equiv^{bubble}\ (\!|\ \phi_S; \mathrm{B}^\circ\ |\!)^{r,t}_C}$$

$$\frac{\exists t \in \phi_T\ bubbleCreated(\phi_T, \sigma_T, \Lambda, t)}{([b], \phi_T, \sigma_T, \Lambda, n) \ \equiv^{bubble}\ \circ}$$

$$\frac{\exists t \in \phi_T\ Close(\phi_T, \sigma_T, \Lambda, t)}{([b], \phi_T, \sigma_T, \Lambda, n) \ \equiv^{bubble}\ \circ}$$

$$([], \phi_T, \sigma_T, \Lambda, n) \ \equiv^{bubble}\ \circ$$

Équivalence d'états L'équivalence de structure vérifie l'équivalence de bulles entre l'état cible est la bulle de plus haut niveau.

L'équivalence de structure est définit de la façon suivante :

$$\frac{\begin{array}{c} \exists xs, listeBulle_{\sigma_T,\Lambda}(getlistBubbleState((\phi_T, \sigma_T, \Lambda)), xs) \wedge \\ xs, \sigma_T, \Lambda, 0 \equiv^{bubble} (\!|\ \phi_S; \mathrm{B}^\circ\ |\!)^{r,t}_C \end{array}}{(\phi_T, \sigma_T, \Lambda) \equiv^{struct} ((\!|\ \phi_S; \mathrm{B}^\circ\ |\!)^{r,t}_C, \sigma_S, r)}$$

L'équivalence entre deux états se base sur l'équivalence de structure, et suppose le renommage des identifiants de processus légers par la fonction *renameTid*() appliqué sur l'état cible.

$$\frac{(\phi_T, \sigma_T, \Lambda) \equiv^{struct} (\!|\, \phi_S; \mathrm{B}^\circ \,|\!)^{r,t}_C, \sigma_S, r) \quad \sigma_T \equiv^{heap} \sigma_S}{(\phi_T, \sigma_T, \Lambda) \equiv (\!|\, \phi_S; \mathrm{B}^\circ \,|\!)^{r,t}_C, \sigma_S, r)}$$

En plus des définition de l'équivalence entre états, nous définissons également l'invariant d'un état accessible $\Sigma = (\phi, \sigma, \Lambda, \mathcal{A})$ depuis un programme compilé $\mathcal{C}(p)$. Celui-ci consiste à s'assurer de la validité de la structure support à la compilation.

$Inv(\Sigma) \equiv$

$$(\forall t \in \phi, \forall b, \rho, tid,\ t = (b, \rho)^{tid},\ \neg ForkCB(\Sigma, t) \Rightarrow isInfo_{\sigma,\Lambda}(\rho(info))) \wedge \tag{9.1}$$

$$(\forall t_1, t_2, \rho_1, \rho_2, b_1, b_2,\ Initialised(\Sigma, t_1) \wedge Initialised(\Sigma, t_2) \wedge t_1 = (b_1, \rho_1) \wedge t_2 = (b_2, \rho_2) \Rightarrow \tag{9.2}$$

$$getlistBubble_\sigma(\rho_1) = getlistBubble_\sigma(\rho_2)) \wedge$$

$$(\forall t, b, \rho, tid,\ t = (b, \rho)^{tid} \wedge (\exists x, m, e, hd(b) = x := \texttt{fork}(m, e) \Rightarrow isInfo_{\sigma,\Lambda}(\sigma(\rho(arg))[0])) \wedge \tag{9.3}$$

$$(\forall t, b, \rho, tid,\ t = (b, \rho)^{tid} \wedge hd(b) = forkCB(0) \Rightarrow isInfo_{\sigma,\Lambda}(\sigma(\rho(arg))[0])) \wedge \tag{9.4}$$

Pour tout bloc de code instrumenté et pour chaque instruction de ce bloc la condition associée à cette instruction doit être vérifiée (9.5)

9.2.2 Préservation sémantique

Nous définissons maintenant les lemmes utiles pour la preuve de la compilation.

Étant donné une réduction d'un état d'un programme vers un autre état, on souhaite avoir certaines informations sur l'instruction active suivante. Cela est utile pour savoir quelles conditions sont vérifiées.

Lemme 9.1 *Étant donné deux états accessibles d'un programme $\mathcal{C}(p)$ Σ et Σ', une action a et un identifiant de processus léger t_0 telle que $\vdash_p \Sigma \xrightarrow{(t_0,a)}_L \Sigma'$, le prédicat suivant renseigne sur certaines propriétés valides dans le nouvel état.*

$\exists t, b, pos,\ S\ pos < |b| \wedge instrOfThread_\phi t = b(pos) \wedge snd(a) = t \Rightarrow$
$fst(\Sigma') = \phi' \wedge instrOfThread_{\phi'} t = b(S\ pos)$

$\exists t, x, fst(\Sigma) = \phi \wedge (\exists m, e,\ instrOfThread_\phi t = x := \texttt{fork}(m, e)) \wedge snd(a) = t \Rightarrow$
$instrOfThread_{\phi'} \rho(x) = forkCB(0)$ *où* $snd(\phi(t)) = \rho$

$\exists t, x, fst(\Sigma) = \phi \wedge instrOfThread_\phi t = forkPB(16) \wedge snd(a) = t \Rightarrow$
$instrOfThread_{\phi'} t = \cdot := \texttt{fork}(\cdot, \cdot)$

Lemme 9.2 *Pour tout état vérifiant l'invariant, un seul processus léger peut être dans un bloc open actif.*

Esquisse de preuve. Nous le prouvons par contradiction, en supposant deux processus légers différents t_1, t_2 dans un bloc *open* actif. Pour pouvoir ouvrir une bulle, un processus léger doit être dans la bulle dont le champ *next* est *null* (la bulle la plus à droite). t_1 et t_2 sont donc nécessairement dans la même bulle. Lorsqu'un processus léger est dans un bloc *open* actif, il a le verrou de sa bulle. Deux processus légers différents ne peuvent pas détenir le même verrou en même temps, l'hypothèse n'est pas vérifiée, la propriété est donc vérifiée. □

Lemme 9.3 *Pour tout état vérifiant l'invariant, un seul processus léger peut être dans un bloc close actif.*

Esquisse de preuve. La preuve est identique au lemme 9.2. □

Lemme 9.4 *Pour tout état Σ_T, Σ_S, tel que $\Sigma_T = (\phi_T, \sigma_T, \Lambda, \mathcal{A})$ et $\Sigma_S = (B, \sigma_S, S, \mathcal{A})$, pour tout processus léger t_T, t_S appartenant aux états, tel que $t_T = (b_T, \rho_T)$, $t_S = (b_S, \rho_S)$ et $b_T, \rho_T, \sigma_T \equiv^{env} \rho_S$, et pour toute expression $e \in hd(b_T) \wedge e \in hd(b_S)$, nous avons soit:*

- *$\exists \ell_s, \ell_T$ tel que $[\![e]\!]_{\rho_S,\sigma_S} = \ell_S$ et $[\![e]\!]_{\rho_T,\sigma_T} = \ell_T$ et $\ell_{S\ \rho_S,\sigma_S,\emptyset} \sim^{add}_{\rho_T,\sigma_T,\emptyset} \ell_T$,*
- *$\exists tid_s, tid_T$ tel que $[\![e]\!] = tid_s$ et $[\![e]\!] = tid_T$ et $\phi_S(tid_S) \equiv^{thread} \phi_T(tid_T)$,*
- *$[\![e]\!]_{\rho_S,\sigma_S} = [\![e]\!]_{\rho_T,\sigma_S}$ sinon.*

Démonstration.
- Si l'expression est une constante, le résultat est immédiat.
- Si l'expression est une variable
  - et que celle-ci contient une constante non adresse, l'équivalence des environnements locaux permet de conclure,
  - et que celle-ci contient une adresse, l'équivalence du tas permet de conclure,
  - et que celle-ci contient un identifiant de processus léger, l'équivalence des ensembles de processus légers permet de conclure.
- Si l'expression contient une opération: L'opération ne peut pas porter sur les adresses car il n'y a pas d'arithmétique de pointeurs, ni sur les identifiants de processus légers.
  - opération sur les entiers : on conclut avec une combinaison des arguments précédents
  - $[\![e_1 = e_2]\!]$: Suivant le résultat de l'évaluation :
    - *true* : On sait par définition de l'égalité sur les adresses que $[\![e_1]\!] = [\![e_2]\!] \wedge [\![e_1]\!] \in dom(\sigma) \wedge [\![e_2]\!] \in dom(\sigma)$. Soit $\ell = [\![e_1]\!]$. Nous considérons deux cas, suivant que cette adresse soit dans la liste d'exclusion $listp_{\sigma_T,\rho_T}$ ou non.
      - $\ell \notin listp_{\sigma_T,\rho_T}$: On sait par le lemme 8.3 que $G_{\sigma_T}^{listp_{\sigma_T,\rho_T}}([\![e_1]\!]) = G_{\sigma_T}^{listp_{\sigma_T,\rho_T}}([\![e_2]\!])$ et $G_{\sigma_T}^{listp_{\sigma_T,\rho_T}}([\![e_1]\!]) \neq \perp$. On sait que comme l'expression s'évalue en une adresse, l'expression est nécessairement une va-

riable, car il n'y a pas de constante adresse. Par définition de l'équivalence on sait donc que $G^{listp_{\sigma_T,\rho_T}}_{\sigma_T}([\![e_1]\!]) = G^{\emptyset}_{\sigma_S}([\![e_1]\!])$ et $G^{listp_{\sigma_T,\rho_T}}_{\sigma_T}([\![e_2]\!]) = G^{\emptyset}_{\sigma_S}([\![e_2]\!])$. Comme l'on sait que $G^{\emptyset}_{\sigma_S}([\![e_1]\!]) \neq \bot$, en appliquant le lemme 8.4, on sait qu'il existe une adresse logique ℓ_1^p telle que $\ell_1^p \in dom(\sigma_S) \wedge G^{\emptyset}_{\rho_S}([\![e_1]\!]) = \ell_1^p$. Nous avons un résultat similaire pour $G^{\emptyset}_{\sigma_S}([\![e_2]\!])$. Avec le lemme 8.3 on sait que $\ell_1 = \ell_2$, et donc $[\![e_1 = e_2]\!] = true$. On a donc bien le même résultat pour l'évaluation.

— $\ell \in listp_{\sigma_T,\rho_T}$: On sait donc par définition du graphe atteignable que $G^{listp_{\sigma_T,\rho_T}}_{\sigma_T}([\![e_1]\!]) = \bot$. Donc par équivalence des graphes, on sait également que $G^{\emptyset}_{\sigma_S}([\![e_1]\!]) = \bot$. Par le lemme 8.5 on sait que $[\![e_1]\!] \notin dom(\sigma_S) \vee [\![e_1]\!] \in \emptyset$. On sait donc que $[\![e_1]\!] \notin dom(\sigma_S)$. Donc l'évaluation de l'expression n'est pas définie, le programme source est bloqué dans un état non final, ce qui est en contradiction avec l'hypothèse $safe(p)$, l'hypothèse $\ell \in listp_{\sigma_T,\rho_T}$ n'est pas possible.

— *false* : La preuve est similaire au cas précédent.

— non définie : On sait que $[\![e_1]\!] \notin dom(\sigma_T)$ ou $[\![e_2]\!] \notin dom(\sigma_T)$ ou les deux. Supposons que $[\![e_1]\!] \notin dom(\sigma_T)$ (le raisonnement est identique pour les autres cas). On sait donc que par définition du graphe atteignable que $G^{listp_{\sigma_T,\rho_T}}_{\sigma_T}([\![e_1]\!]) = \bot$. On sait aussi par équivalence des graphes que $G^{\emptyset}_{\sigma_S}([\![e_1]\!]) = \bot$. Donc par le lemme 8.5 on sait que $[\![e_1]\!] \notin dom(\sigma_S)$. Donc l'évaluation de l'expression n'est pas définie, le programme source est bloqué dans un état non final, ce qui est en contradiction avec l'hypothèse $safe(p)$. L'hypothèse de l'évaluation de l'égalité non définie n'est pas possible.

□

Lemme 9.5 *Pour tout bloc d'instruction b_1, b2,instructions c_1, c_2 et environnements locaux ρ_1, ρ_2, si $c_1|\blacksquare; b_1 \equiv^{block} c_2; b_2$ alors $b_1 \equiv^{block} b_2$.*

Esquisse de preuve. La preuve se fait par induction sur b_1. □

Lemme 9.6 *Pour tout bloc d'instruction b_1, b2,instructions c_1, c_2 et environnements locaux ρ_1, ρ_2, si $c_1|\blacksquare; b_1 \equiv^{block} c_2; b_2$ alors $b_1 \equiv^{block} c_2; b_2$.*

Esquisse de preuve. La preuve se fait par induction sur b_1. □

Le résultat de la fonction *filterVar*() est identique avant et après l'exécution d'une instruction `mlock` qui ne contient que des lectures dans des variables réservées.

Lemme 9.7 *Soit un état Σ_T d'un programme compilé, et soit une liste d'instruction b qui ne comporte que des lectures dans des variables réservées et une expression e. Soit Σ_T' l'état obtenu après l'exécution de l'instruction* `mlock` *b e. On a filterVar(ρ) = filterVar(ρ') (où ρ et ρ' sont les environnements locaux des deux états respectifs Σ_T et Σ_T').*

Esquisse de preuve. L'instruction modifie uniquement des variables réservées. Par définition de *filterVar*(), les variables réservées ne sont pas prises en compte. La modification de telles variables n'a dont pas d'effet sur le résultat de la fonction. □

Lemme 9.8 *Pour tout état atteignable d'un programme compilé, il n'y a pas d'instruction* `mlock`, `lock` *ou* `unlock` *dans une instruction non instrumentée.*

Esquisse de preuve. Un programme source est constitué des instructions non instrumentées issues du programme source, sans modifications, et des instructions instrumentées. Dans la syntaxe du langage du programme source, les instructions `mlock`, `lock` ou `unlock` n'existent pas. Il n'y a donc pas de telles instructions dans les instructions non instrumentées du programme compilé. □

Pour tout programme compilé, il n'existe pas de *close* pendant dans un bloc complet de méthode.

Lemme 9.9 *Pour tout programme compilé $\mathcal{C}(p)$, pour toute méthode m de ce programme, et tout bloc b_m qui est le corps de la méthode m, alors $prefixPendingClose(b_m) = []$.*

Esquisse de preuve. La preuve s'effectue grâce à la définition de la compilation. Si le bloc de la méthode du programme source ne contient aucune section, le résultat est immédiat. S'il existe une section, celle-ci se traduit par un bloc *open* et un *close*, donc il n'y a pas de bloc pendant. □

Nous commençons la preuve en inspectant les instructions non instrumentées. On sait grâce au lemme 9.8 qu'il n'y a pas d'instruction `mlock`, `lock` ou `unlock` dans le code non instrumenté, nous ne considérons donc pas la règle de sémantique associée. t_T est le processus léger actif. Pour l'équivalence des tas, comme l'instruction que nous considérons est non instrumentée, nous ne prenons en compte que la dernière propriété.

Proposition 9.1 *Étant donné deux états Σ_T et Σ_S et un programme source p tel que $safe(p)$, une action a et un identifiant de processus léger t, tel que $\Sigma_T \sim \Sigma_S$, $Inv(\Sigma_T)$ et $\vdash_{\mathcal{C}(p)} \Sigma_T \xrightarrow{(t,a)}_L \Sigma_T'$ alors :*

- *$Inv(\Sigma_T')$*
- *et soit*
  - *l'action a n'est pas colorée, et il existe Σ_S' tel que $\vdash_p \Sigma_S \xrightarrow{(t,a)}_A \Sigma_S'$ et $\Sigma_T' \sim \Sigma_S'$*
  - *l'action a est colorée et l'on a $|\Sigma_T'| < |\Sigma_T|$ et $\Sigma_T' \sim \Sigma_S$.*

Esquisse de preuve.

- (**assign**) $x := e$
  - $Inv(\Sigma_T')$ L'invariant est conservé car ni le tas, ni les variables réservées ne sont modifiées. Si la prochaine instruction est une instruction instrumentée différente de la première de *forkCB* , il n'y a pas de conditions supplémentaire. Dans le cas de *forkCB* la condition est prise en compte dans l'invariant.

- $\Sigma_T^{'} \sim \Sigma_S^{'}$:
  - $\Sigma_T^{'} \equiv^{struct} \Sigma_S^{'}$: On sait que $\exists xs, listeBulle_{\sigma_T,\Lambda}(getlistBubbleState((\phi_T, \sigma_T, \Lambda)), xs)$. $xs, \sigma_T, \Lambda, 0 \equiv^{bubble} (\!| \phi_S; \mathrm{B}^\circ |\!)_C^{r,t}$
    - $listPointerListInfo_{\sigma,\Lambda}(first), \phi_T, \sigma \equiv^{threads} \phi_S$: Seul le processus léger t_T est modifié, les autres processus ne le sont pas, l'équivalence est donc conservée dans leur cas. Soit t_S le processus léger équivalent à t_T dans l'état source. D'après le lemme 9.5 l'équivalence d'instructions est conservée. D'après le lemme 9.4, on sait que l'expression s'évalue de façon similaire dans les deux états. Si elle s'évalue en une valeur non adresse la valeur est identique dans les deux états. Donc l'équivalence des environnements locaux est conservée. Si l'expression s'évalue en une adresse alors elle n'entre pas en compte dans l'équivalence des états. Pour le reste des preuves, nous ne considérerons que la preuve de l'équivalence entre le processus léger actif t_T et son équivalent dans l'état source.
    - $t_T^o \equiv^{thread} t_S^o$: Si $t_T = t_T^o$ alors nous déduisons l'équivalence du point précédent. Sinon l'équivalence est conservée. La preuve étant identique pour tous les autres cas, elle ne sera pas répétée.
    - $blockClose(t_T^o, n - |xs'| + 2) \equiv^{block} C$:Si $t = t_T^o$ alors l'instruction est nécessaire à droite du bloc retourné s'il existe, l'équivalence n'est donc pas modifié. Si $t \neq t_T^o$ alors l'équivalence n'est pas modifiée. La preuve étant identique pour tous les autres cas, elle ne sera pas répétée.
    - $xs', \phi_T, \sigma_T, \Lambda, S\ n \equiv^{bubble} \mathrm{B}^\circ$: Les autres processus légers ne sont pas modifiés, l'équivalence est conservée.
  - $\Sigma_T^{'} \sim^{\text{heap}} \Sigma_S^{'}$: Le tas n'est pas modifié. La modification de la variable peut supprimer sa présence dans les adresses racines (dans le cas ou la variable contenait une adresse et ensuite une valeur non adresse). Dans ce cas, l'ensemble des valeurs calculées par le graphe mémoire diminue. Si l'expression s'évalue en une adresse, on sait, d'après le lemme 9.4, que l'adresse évaluée dans le programme compilé est équivalente à celle évaluée dans le programme source. D'après le même lemme si l'expression s'évalue en un identifiant de processus léger, on sait également que les identifiants des processus légers sont équivalents. Les deux graphes mémoires restent équivalents.
- (**alloc**) $x := \texttt{allocate}(e)$
  - $Inv(\Sigma_T^{'})$ Le tas est modifié, mais les structures de bulles et d'infos ne sont pas modifiées. L'invariant est donc toujours valide.
  - $\Sigma_T^{'} \sim \Sigma_S^{'}$:
    - $\Sigma_T^{'} \equiv^{struct} \Sigma_S^{'}$:
      - $t_T \equiv^{thread} t_S$: D'après le lemme 9.5 l'équivalence d'instructions est conservée. L'allocation place une adresse dans la variable x. Donc si x contenait une valeur non adresse avant, celle-ci n'est plus prise en

compte dans les deux environnements locaux, cible et source, l'équivalence est conservée. Si x contenait déjà une adresse ou n'était pas présent dans $dom(\rho_T)$, l'équivalence n'est pas modifiée.
  - $\Sigma_T^{'} \sim^{\text{heap}} \Sigma_S^{'}$: On peut conclure grâce aux lemmes 8.1 et 8.2
- (**dispose**) `dispose`(e)
  - $Inv(\Sigma_T^{'})$ On sait que l'adresse correspond à un élément également présent dans le tas source (voir preuve de $\Sigma_T^{'} \sim^{\text{heap}} \Sigma_S^{'}$). La modification du tas n'a pas donc d'impact sur les structures de bulles et d'*infos* existantes. L'invariant est conservé.
  - $\Sigma_T^{'} \sim \Sigma_S^{'}$:
    - $\Sigma_T^{'} \equiv^{struct} \Sigma_S^{'}$:
      - $t_T \equiv^{thread} t_S$: D'après le lemme 9.5 l'équivalence d'instructions est conservée. D'après la règle (**dispose**), l'environnement local n'est pas modifié, l'équivalence d'environnements est donc conservée.
    - $\Sigma_T^{'} \sim^{\text{heap}} \Sigma_S^{'}$: On sait d'après la règle **dispose** que e s'évalue en une adresse. On sait d'après le lemme 9.4 que les adresses sont équivalentes. On peut donc conclure par application du lemme 8.2.
- (**get**)$x := y[e]$
  - $Inv(\Sigma_T^{'})$ Le tas n'est pas modifié, ni les variables réservées, donc l'invariant reste valide.
  - $\Sigma_T^{'} \sim \Sigma_S^{'}$:
    - $\Sigma_T^{'} \equiv^{struct} \Sigma_S^{'}$:
      - $t_T \equiv^{thread} t_S$: D'après le lemme 9.5 l'équivalence d'instructions est conservée. On sait d'après la règle (**get**) que e s'évalue nécessairement en un entier n. D'après l'équivalence de tas $\Sigma_T \sim^{\text{heap}} \Sigma_S$ cet entier est le même dans les états source et cible. Si $y[n]$ s'évalue en une valeur non adresse, d'après l'équivalence de tas, cette valeur est identique dans les deux états, donc la modification est identique pour les deux environnements locaux. L'équivalence est donc conservée. Si $y[n]$ s'évalue en une adresse dans l'état cible, d'après l'équivalence des tas, $y[n]$ s'évalue également en une adresse dans l'état source. Les variables x dans les deux états reçoivent donc toutes les deux une adresse. Les variables contenant une adresse ne sont pas prises en compte dans l'équivalence des environnements locaux, l'équivalence est conservée.
    - $\Sigma_T^{'} \sim^{\text{heap}} \Sigma_S^{'}$: Le tas n'est pas modifié, et d'après le point précédent l'équivalence des environnements locaux est conservée. L'équivalence des tas est donc également conservée.
- (**put**) $x[e] := y$
  - $Inv(\Sigma_T^{'})$ La modification du tas ne modifie pas les structure de bulles et d'infos existantes, voir la preuve de $\Sigma_T^{'} \sim^{\text{heap}} \Sigma_S^{'}$. L'équivalence est toujours valide.
  - $\Sigma_T^{'} \sim \Sigma_S^{'}$:

  - $\Sigma_T^{'} \equiv^{struct} \Sigma_S^{'}$:
    - $t_T \equiv^{thread} t_S$: : D'après le lemme 9.5 l'équivalence d'instructions est conservée. D'après la règle (**put**) l'environnement local n'est pas modifié, l'équivalence est donc conservée.
  - $\Sigma_T^{'} \sim^{\text{heap}} \Sigma_S^{'}$: S'il existe une adresse ℓ_1, telle que $\rho_T(y) = \ell_1$ alors il existe une adresse ℓ_2, telle que $\rho_S(y) = \ell_2$ et d'après $\Sigma_T \sim^{\text{heap}} \Sigma_S$, on sait que l'adresse ℓ_1 est équivalente à ℓ_2. La modification du tas conserve la relation d'équivalence. Si $\exists v\ \rho_T(y) = v$ où v est une valeur non adresse, alors $\rho_S(y) = v$ d'après l'équivalence $\Sigma_T \sim^{\text{env}} \Sigma_S$. L'équivalence des tas est également conservée.
- (**fork**) $x := \texttt{fork}(m, e)$
  - $Inv(\Sigma_T^{'})$ D'après le lemme, la prochaine instruction du processus léger présente $\rho(x)$ est la première instruction de *forkCB*. On sait d'après l'invariant que $\sigma(arg)(0)$ est bien une structure *info*. Donc on sait que pour le nouveau processus léger, dans le prochain état, nous aurons bien $isInfo_{\sigma,\Lambda}(\sigma(\rho_c(arg))(0))$ où ρ_c est l'environnement local du nouveau processus léger car *arg* est l'argument donnée à la création du processus léger.
  - $\Sigma_T^{'} \sim \Sigma_S^{'}$:
    - $\Sigma_T^{'} \equiv^{struct} \Sigma_S^{'}$:
      - $t_T \equiv^{thread} t_S$: D'après le lemme 9.5 l'équivalence d'instructions est conservée. La variable x reçoit l'identifiant du nouveau processus léger. Il n'est donc pas pris en compte dans l'équivalence des environnements locaux. L'équivalence est donc conservée.
      - $listPointerListInfo_{\sigma,\Lambda}(first), \phi_T, \sigma \equiv^{threads} \phi_S$: Un nouveau processus léger est créé dans chaque état, soit t_T^n, t_S^n. La méthode appelée est identique dans les deux cas. Soit b_m le corps de la méthode. Pour le processus léger t_T^n le corps de la méthode est $\mathcal{C}(b_m)$, et pour t_S^n c'est b_m. On sait que $\mathcal{C}(b_m) = forkCB; b_m; forkCE$. D'après le lemme 9.9, on sait qu'il n'y a pas de bloc *close* pendant, donc $prefixPendingClose(forkCB; b_m; forkCE) = forkCB; b_m; forkCE$ On sait également qu'aucun bloc d'instructions instrumentées n'est entamé, et donc $removePrefixSuffix(forkCB; b_m; forkCE) = forkCB; b_m; forkCE$. On a donc bien l'équivalence des blocs.
      - équivalence des environnements des nouveaux processus légers : L'instruction en tête du nouveau processus léger dans l'état cible est la première de *forkCB*. On sait que dans $\sigma(arg)(1)$ est contenu l'évaluation de l'argument original. Les environnements des nouveaux processus légers sont donc équivalents.
    - $\Sigma_T^{'} \sim^{\text{heap}} \Sigma_S^{'}$: D'après la règle (**fork**), le tas n'est pas modifié. L'environnement local est modifié par l'écriture dans la variable x. Mais on sait que les environnements locaux restent équivalent d'après le premier point. Donc le tas reste équivalent pour ces processus légers. Nous devons également prouver l'équivalence du tas

à partir des nouveaux processus légers. Soit t_2, le nouveau processus léger dans LUFJ. Alors la condition $ForkCB(\Sigma, t_T)$ est vérifiée. L'équivalence de tas est donc l'équivalence des graphes atteignables suivante : $reachableGraph_{\sigma_T}(\sigma(\rho(arg))(1), listeExclusion_{\sigma_T}(\rho_T) \cup \rho(childInfo)) = reachableGraph_{\sigma_S}(\rho_S, \emptyset))$. On sait que dans $\rho(arg))(1)$ contient le résultat de l'évaluation de l'expression original de l'appel de la création de processus léger. L'environnement local source contient également uniquement une expression contenant cette valeur. Les deux graphes atteignables sont donc équivalents.

— (**join**) `join` e
  — $Inv(\Sigma_T^{'})$ Le tas, l'environnement local ne sont pas modifiés. L'invariant reste valide.
  — $\Sigma_T^{'} \sim \Sigma_S^{'}$:
    — $\Sigma_T^{'} \equiv^{struct} \Sigma_S^{'}$:
      — $t_T \equiv^{thread} t_S$: D'après le lemme 9.5 l'équivalence d'instructions est conservée. D'après la règle (**put**) l'environnement local n'est pas modifié, l'équivalence est donc conservée.
      — $listPointerListInfo_{\sigma,\Lambda}(first), \phi_T, \sigma \equiv^{threads} \phi_S$: D'après la règle (**join**), on sait que $[\![e]\!]$ s'évalue en un identifiant de processus léger, soit t_T^j. Elle s'évalue également en un tel identifiant dans l'état source, soit t_S^j. D'après le lemme 9.4 on sait le processus léger désigné est équivalent, c'est à dire que $\phi_T(t_T^j) \equiv^{thread} \phi_S(t_S^j)$. L'équivalence des autres processus légers est donc conservée.
    — $\Sigma_T^{'} \sim^{\text{heap}} \Sigma_S^{'}$: Le tas n'est pas modifié. L'expression e s'évalue en un processus léger dans les deux états d'après les règles de sémantique. Ces deux processus légers sont équivalents d'après le lemme 9.4. La suppression est donc similaire dans les deux ensembles de processus légers. L'équivalence du tas est donc conservée.

Nous nous intéressons maintenant aux instructions instrumentées. Nous les examinons par bloc. Ce qui suit reste très informel. Une preuve complète nécessite de prouver des propriétés avant chaque instruction du bloc. Il manque également une mesure d'un état. Comme l'invariant est souvent invalide pendant ces blocs, ces propriétés permettent de prouver l'invariant en sortie de bloc. Une telle preuve est visible dans le chapitre 8.

— *open* Prouver l'équivalence après chaque instruction de ce bloc consiste à prouver cette relation entre l'état cible $\Sigma_T^{'}$ vis à vis de l'état source de départ Σ_S. Ce bloc crée une nouvelle bulle et déplace le *info* du processus léger propriétaire. Le but est donc de tenir compte de cette création et de ce déplacement dans l'équivalence. Pour l'équivalence de structure, une fois la nouvelle bulle ajoutée à la liste des bulles, il faut ne pas la prendre en compte dans l'équivalence, car nous prouvons l'équivalence avec l'état source Σ_S, où la nouvelle section n'est

pas créée. C'est grâce à la deuxième équivalence de structure et la condition *bubbleCreated*.
La structure *info* associée au processus léger qui est dans ce bloc est déplacée dans la nouvelle bulle. Suivant le même principe que pour la nouvelle bulle, il faut considérer pour l'équivalence que la structure *info* n'a pas été déplacée. Ceci est possible grâce à la définition et l'équivalence et de la condition *RemoveInfo* qui est vérifiée dans ce cas.
L'équivalence du tas est également particulière. La nouvelle bulle est créée mais pendant quelques instructions (1 à 4) elle n'est pas ajoutée à la liste de bulle, qui est une partie de la liste d'exclusion. Ce cas est prévu dans la définition de l'équivalence de tas lorsque la condition *Open* est vérifiée.
L'invariant est valide après le bloc car l'ajour de la nouvelle bulle conserve la structure de bulle. Il y a toujours une structure *info* dans le processus léger responsable de l'ouverture.

— *close* La preuve de l'équivalence se déroule en deux parties. Pour les instructions dont le numéro est inférieur à 4, l'équivalence de l'état cible $\Sigma_T^{'}$ se fait avec l'état source Σ_S, autrement dit, on avance dans le programme cible, mais pas dans le programme source. Ensuite les processus légers sont déplacés de la bulle en train d'être fermée vers la précédente. La difficulté se situe dans la différence entre ce déplacement dans le programme source et le programme compilé. Dans le premier, l'ensemble des processus légers est déplacé en une étape de réduction, alors que dans le programme compilé, chaque processus léger est déplacé individuellement, donc en plusieurs étapes de réduction. C'est pourquoi pour l'instruction 4, nous considérons l'équivalence de structure entre l'état source $\Sigma_S^{'}$ où l'action *close* a été effectuée et l'état cible modifié : nous considérons que tous les processus légers de la dernière bulle sont dans l'avant dernière, et nous ne prenons plus en compte la dernière bulle. Pour les instructions suivantes de ce bloc, nous considérons toujours cet état cible modifié, mais l'état cible est équivalent à l'état source Σ_S.
Les structures *info* déplacées sont toujours valides après l'exécution du bloc.

— *initialize* Dans ce bloc, on retrouve les blocs *newBubble* et *newInfo*, ainsi que des écritures dans les emplacements mémoires de ces nouvelles structures. Nous prouvons l'équivalence entre l'état cible $\Sigma_T^{'}$ après chaque instruction, et l'état source de départ Σ_S. Pour les blocs *newBubble* et *newInfo*, nous avons vu qu'ils ne modifient pas l'équivalence. L'écriture ne modifient pas non plus l'équivalence car elles modifient des variables réservées.
L'invariant est valide après le bloc car la variable *info* est initialisé avec la nouvelle bulle.

— *forkPB* : Ce bloc d'instructions crée la nouvelle structure *info* représentant le nouveau processus léger. Ce bloc est placé juste avant l'instruction `fork`, nous considérons donc l'équivalence après chaque instruction de ce bloc entre le nouvel état cible $\Sigma_T^{'}$ et l'état source de départ. Il est nécessaire de prendre en compte le nouvel *info* qui a été créé pour l'équivalence de structure. À partir

de l'instruction 6, ce nouvel élément est ajouté à la liste des *info*. Comme nous sommes avant l'instruction `fork`, il ne faut pas le prendre en compte. C'est pourquoi dans la définition de l'équivalence, il est ignoré.
D'après le lemme 9.1, la prochaine instruction de ce processus léger est `fork` . La variable *childInfo* contient bien une structure *info* car celle-ci à été initialisé avec le bloc *newInfo*. Les autres conditions de l'invariant restent valides.

— *forkCB* : D'après la propriété, on sait que $arg[1] = e$, où e est l'argument original. Après chaque instruction du bloc (sauf la dernière), on prouve l'équivalence entre l'état cible $\Sigma_T^{'}$ et l'état source de départ Σ_S.
On sait l'invariant, en début de bloc que la variable $arg[0]$ contient bien une structure *info*. Du fait de l'affectation à la variable *info*, l'invariant est valide à la fin du bloc.

— *forkCE* : Ce bloc est placé après les instructions du corps d'une méthode. Donc on prouve l'équivalence de l'état cible $\Sigma_T^{'}$ après chaque instruction avec celui de l'état source de départ Σ_S. Le bloc consiste à enlever la structure *info* correspondant au processus léger qui vient de terminer de la liste des processus légers de la bulle, et de le désallouer. Nous considérons l'équivalence entre l'état source ou le processus léger a terminé, donc l'équivalence d'instruction est immédiate. L'équivalence de tas est immédiate car *info* fait parti de la liste des *info*, sauf lorsqu'il est détaché de la liste, mais pas encore désalloué. C'est pourquoi, l'équivalence de tas prend en compte cette variable en l'ajoutant à la liste d'exclusion grâce à la condition *RemoveInfo*. L'équivalence des environnement locaux est immédiate.
L'invariant reste valide.

— *newInfo* : Ce bloc a pour but de créer une nouvelle structure *info* dans la variable *info* du processus léger. Il contient une allocation mémoire, une affectation, et des écritures. L'allocation se fait dans la variable *info*. Ce cas est traité par *forkPB* et *initialize* où il est appelé. Les écritures se font ensuite dans ces cellules mémoires. Comme elles ne sont pas prises en compte dans le graphe mémoire, cela ne modifie pas le graphe atteignable. L'affectation se fait également sur une variable réservée, elle n'a donc également pas d'impact sur l'équivalence des environnement locaux.

— *newBubble* : Ce cas est traité dans les blocs *open* et *initialize*.

— *remove* : Le but de ce bloc est de supprimer la variable *info* du processus léger, et de la liste des *info* de la bulle. Par les écritures du tas, seuls des éléments de cette liste sont modifiés. Cette liste fait partie de la liste d'exclusion du calcul du graphe atteignable, celui-ci n'est donc pas modifiée par ces écritures. Seules des variables réservées sont modifiées par des affectations, l'équivalence de tas n'est donc pas modifiée. Les prises de verrous ne modifient pas l'équivalence.

— *removeFirst* : Le but de ce bloc est de supprimer le premier élément *info* d'une liste d'une bulle. Comme pour le bloc *remove*, les écritures se font dans des variables appartenant à la liste d'exclusion, le graphe atteignable n'est donc pas

modifié. Les affectations ont lieu dans des variables réservées, qui ne modifient donc pas l'équivalence des environnements locaux.

□

Conclusion et perspectives 10

Sommaire

10.1 Bilan

Les architectures parallèles sont désormais communes. Cependant il est difficile avec les langages de programmations actuels de les exploiter facilement, la faute notamment au manque d'abstractions pour gérer la mémoire partagée. Cependant, de nouvelles propositions émergent, parmi lesquels les transactions. Nous nous sommes intéressés à la notion plus abstraite sous-jacente de ce modèle, les sections atomiques. Nous avons considéré des programmes impératifs capables de créer des sections atomiques emboîtées et supportant le parallélisme interne. L'emboîtement des sections atomiques est une caractéristique importante pour la modularité des programmes. Les processus créés à l'intérieur d'une section ne doivent pas se synchroniser avec la fin de la section, contrairement à la condition habituelle des autres propositions de langages à sections atomiques.

Nous avons tout d'abord défini précisément la notion d'atomicité. Pour cela nous n'avons pas considéré de langage précis, mais nous nous sommes basés sur des traces de programmes. Nous avons énoncé des propriétés de bonnes formations sur les traces pour nous abstraire de la syntaxe d'un programme, mais en éliminant les traces qui ne peuvent pas correspondre raisonnablement à l'exécution d'un programme. Parmi ces conditions nous supposons l'atomicité faible : des sections concurrentes ne peuvent pas s'entrelacer. Nous avons défini dans ce contexte la notion de bonne synchronisation et montré qu'elle implique la notion d'atomicité forte. Ces résultat ont été formalisés dans l'assistant de preuve Coq.

Nous avons ensuite considéré cet ensemble de conditions de bonne formation des traces comme des spécifications de sémantiques opérationnelles de langages de programmation impératifs avec sections atomiques emboîtées et parallélisme interne. Nous avons proposé un tel langage que nous avons appelé AFJ. Nous avons décrit sa sémantique opérationnelle. Un point important est la définition d'une partie de l'état d'un programme : la bulle, mécanisme pour assurer l'atomicité. Nous avons ensuite

prouvé que cette sémantique vérifie les conditions de bonne formation. Ce travail a été partiellement modélisé en Coq.

LUFJ est un second langage identique au précédent, excepté pour les primitives de synchronisations où les sections atomiques ont été remplacées par des verrous. Ce mécanisme de synchronisation est plus difficile à manipuler par l'utilisateur que les sections atomiques, mais peut plus facilement être implantable. LUFJ est le langage cible que nous considérons pour la compilation de AFJ.

Nous avons proposé un processus de compilation de AFJ vers LUFJ. Nous avons détaillé les structures nécessaires et les opérations associées qui servent de support à l'exécution. Nous avons deux types des structures représentant les bulles et les processus légers. Nous savons grâce à cela quel est l'emboîtement des sections, et quelle est la section dans lequel s'exécute un processus. Du code supplémentaire est ajouté pour maintenir cette structure cohérente lors de l'ouverture de sections, de création de processus légers, *etc.*

Nous nous sommes ensuite intéressés à la correction de ce schéma de compilation. Nous l'avons fait en deux temps afin de séparer les difficultés. La compilation introduit des variables, des allocations mémoires et du code supplémentaires. Nous avons considéré tout d'abord un noyau commun aux langages AFJ et LUFJ, sans parallélisme, ni mécanisme de synchronisation, afin de nous concentrer sur les mécanismes de gestion de la mémoire. Nous avons prouvé la correction d'une transformation source-à-source pour ce langage. Enfin nous avons abordé la preuve de préservation de la sémantique lors de la phase de compilation de AFJ vers LUFJ. La préservation sémantique n'est pas encore complètement établie. Il faut néanmoins noter que le travail proposé ici est le premier qui à notre connaissance s'attaque à la vérification d'une phase de compilation dans laquelle les primitives de parallélisme changent. Dans CompCertTSO, qui est la proposition la plus achevée pour la compilation d'un langage parallèle, les primitives de parallélisme des langages sources sont essentiellement les mêmes que les primitives des langages cibles.

10.2 Perspectives

Les preuves sur papier de préservation sémantique, même si elles sont un préalable pour construire le raisonnement, montrent leurs limites rapidement. Elles sont en effet longues et fastidieuses, et peuvent donc contenir des erreurs. Un premier objectif est donc de poursuivre la formalisation de notre cadre en Coq et de parvenir à des preuves complètement vérifiées de préservation sémantique de la passe de compilation de AFJ vers LUFJ.

Outre les preuves en elles-mêmes, la formalisation de certains éléments nécessaires s'avère délicate. Par exemple la fonction récursive de calcul du tas symbolique n'est pas une fonction structurellement récursive. Il est donc nécessaire de fournir une preuve de terminaison de cette fonction. L'argument est que l'on examine toujours plus de nouvelles adresses mémoires au cours du parcours, et que le domaine du tas

est fini. Notons au passage que la formalisation du tas en tant que fonction nécessite que cette hypothèse de finitude soit ajoutée, et que sont invariance au cours de l'exécution soit préservée, ou bien que la modélisation du tas soit modifiée et utilise une structure finie. Un autre exemple est la fonction de décompilation du code utilisée dans la définition de l'équivalence d'états. On pourrait penser qu'elle est facilement structurellement récursive, le problème est que Coq décompose les motifs ayant plusieurs constructeurs ou plusieurs occurrences du même constructeur en filtrage successifs, et le code obtenu ne peut plus alors être vérifié structurellement récursif par Coq.

Une fois une preuve complètement achevée en Coq, il pourrait être intéressant de s'intéresser à des passes de compilation supplémentaires : d'une part la compilation de `mlock` en utilisant des constructions plus primitives telles que `trylock`. Cette idée est évoqué dans le chapitre 6, où est décrit le processus de compilation. Cependant cela introduit de l'attente active, pour la prise de verrous. Afin d'éviter ce problème, une passe de compilation alternative à envisager serait de considérer comme langage cible, un langage possédant d'autres mécanismes de plus bas niveau, comme des signaux.

À plus long terme, d'autres schémas de compilation, ou des optimisations préalables ou postérieures à cette phase de compilation mettant en jeu des analyses statiques pourraient être étudiés, à partir du moment où l'on relâche l'hypothèse de bonne formation ($\mathbf{wf}_3$). Ceci nécessiterait bien sûr de modifier les preuves existantes du chapitre 4. Enfin nous pourrions prendre en compte un langage séquentiel plus riche que celui que nous considérons, et intégrer ces primitives et phases de compilations comme des extensions de CompCert.

Annexes

Annexe

A

Sommaire

A.1 Code séquentiel

initProgram

initProgram::=
arg:=**allocate**(2)
arg[0]:=`null`;
arg[1]:=0

callMethod

prepareCallMethod(e) ::=

mExpr := e;
arg[1]:= *mExpr*;

initMethod

initMethod(x) :=

x:=*arg*[1];

add

addList(x, e) ::=
node := **allocate**(3);
size:=e;
node[*F_ADD*] := x;

```
node[F_SIZE] := size;
node[F_NEXT] := arg[0];
arg[0] := node
```

remove

```
removeList(e)::=
current:=arg[0];
addrRm:=e;
while(current[F_ADD]!=addrRm){
    oldCurrent :=current;
    current:=current[F_NEXT];
}

(*current est en tete*)
if (arg[0] = current) then
  arg(0] := current[F_NEXT]
else
  oldCurrent[F_NEXT] := current[F_NEXT]

dispose(current);
```

A.2 Preuves détaillées

À venir

Bibliographie

[1] M. Abadi, A. Birrell, T. Harris, and M. Isard. Semantics of transactional memory and automatic mutual exclusion. In *POPL*, pages 63–74. ACM, 2008. Cité page 33.

[2] A.-R. Adl-Tabatabai, B. T. Lewis, V. Menon, B. R. Murphy, B. Saha, and T. Shpeisman. Compiler and runtime support for efficient software transactional memory. *SIGPLAN Not.*, 41(6):26–37, June 2006. Cité page 30.

[3] Y. Afek, G. Korland, and A. Zilberstein. Lowering STM overhead with static analysis. In K. Cooper, J. Mellor-Crummey, and V. Sarkar, editors, *Languages and Compilers for Parallel Computing*, volume 6548 of *LNCS*, pages 31–45. Springer, 2011. Cité page 31.

[4] K. Agrawal, J. T. Fineman, and J. Sukha. Nested parallelism in transactional memory. In *PPoPP*, pages 163–174. ACM, 2008. Cité page 30.

[5] G. Almasi. PGAS (partitioned global address space) languages. In D. Padua, editor, *Encyclopedia of Parallel Computing*, pages 1539–1545. Springer, 2011. Cité page 4.

[6] E. Ayguadé, N. Copty, A. Duran, J. Hoeflinger, Y. Lin, F. Massaioli, X. Teruel, P. Unnikrishnan, and G. Zhang. The Design of OpenMP Tasks. *IEEE Transactions on Parallel and Distributed Systems*, 20:404–418, 2009. Cité page 3.

[7] L. Baduel, F. Baude, D. Caromel, A. Contes, F. Huet, M. Morel, and R. Quilici. Programming, Deploying, Composing, for the Grid. In J. Cunha and O. F. Rana, editors, *Grid Computing: Software Environments and Tools*. Springer, 2006. Cité page 3.

[8] G. Barthe, D. Demange, and D. Pichardie. Formal verification of an ssa-based middle-end for compcert. *ACM Trans. Program. Lang. Syst.*, 36(1):4:1–4:35, Mar. 2014. Cité page 34.

[9] Y. Bertot and P. Castéran. *Interactive Theorem Proving and Program Development*. Springer, 2004. Cité pages 9 et 50.

[10] A. Bieniusa and P. Thiemann. Proving isolation properties for software transactional memory. In G. Barthe, editor, *ESOP*, volume 6602 of *LNCS*, pages 38–56. Springer, 2011. Cité page 31.

[11] S. Blazy and X. Leroy. Mechanized semantics for the Clight subset of the C language. *Journal of Automated Reasoning*, 43(3):263–288, 2009. Cité page 34.

[12] S. Boldo, J.-H. Jourdan, X. Leroy, and G. Melquiond. A formally-verified C compiler supporting floating-point arithmetic. In *IEEE Symposium on Computer Arithmetic*, pages 107–115. IEEE Computer Society, 2013. Cité page 34.

[13] P. Brinch Hansen. In P. Brinch Hansen, editor, *The origin of concurrent programming*, chapter The invention of concurrent programming, pages 3–61. Springer, 2002. Cité page 27.

[14] D. Caromel and L. Henrio. *A Theory of Distributed Objects*. Springer, 2004. Cité page 3.

[15] C. Cascaval, C. Blundell, M. Michael, H. W. Cain, P. Wu, S. Chiras, and S. Chatterjee. Software transactional memory: Why is it only a research toy ? *Commun. ACM*, 51(11):40–46, 2008. Cité page 5.

[16] B. L. Chamberlain, D. Callahan, and H. P. Zima. Parallel Programmability and the Chapel Language. *International Journal of High Performance Computing Applications*, 21(3):291–312, 2007. Cité page 4.

[17] B. Chapman, T. Curtis, S. Pophale, S. Poole, J. Kuehn, C. Koelbel, and L. Smith. Introducing OpenSHMEM: SHMEM for the PGAS community. In *Proceedings of the Fourth Conference on Partitioned Global Address Space Programming Model*. ACM, 2010. Cité page 4.

[18] A. Chlipala. An Introduction to Programming and Proving with Dependent Types in Coq. *Journal of Formalized Reasoning*, 3(2), 2010. Cité page 9.

[19] A. Chlipala. A verified compiler for an impure functional language. In M. V. Hermenegildo and J. Palsberg, editors, *POPL*, pages 93–106. ACM, 2010. Cité page 34.

[20] A. Chlipala. *Certified Programming with Dependent Types*. MIT Press, 2014. Cité page 50.

[21] M. Cole. *Algorithmic Skeletons: Structured Management of Parallel Computation*. MIT Press, 1989. Available at `http://homepages.inf.ed.ac.uk/mic/Pubs`. Cité page 3.

[22] T. Coquand and G. Huet. The Calculus of Constructions. *Information and Computation*, 37(2-3), 1988. Cité page 9.

[23] Cray Inc. Chapel Language Specification Version 0.93, April 2013. `http://chapel.cray.com/spec/spec-0.93.pdf`. Cité page 4.

[24] D. Cunningham, S. Drossopoulou, and S. Eisenbach. Lock Inference Proven Correct. In *Formal Techniques for Java-like Programs*, July 2008. Cité page 32.

[25] D. Cunningham, K. Gudka, and S. Eisenbach. Keep Off The Grass: Locking the Right Path for Atomicity. In *Compiler Construction 2008*, volume 4959 of *LNCS*, pages 276–290, April 2008. Cité page 32.

[26] P. Damron, A. Fedorova, Y. Lev, V. Luchangco, M. Moir, and D. Nussbaum. Hybrid transactional memory. In *ASPLOS XII*, pages 336–346. ACM, 2006. Cité page 29.

[27] Z. Dargaye. *Vérification formelle d'un compilateur optimisant pour langages fonctionnels*. PhD thesis, Université Paris 7 Diderot, July 2009. Cité page 34.

[28] Z. Dargaye and X. Leroy. A verified framework for higher-order uncurrying optimizations. *Higher-Order and Symbolic Computation*, 22(3):199–231, 2009. Cité page 34.

[29] M. A. Dave. Compiler verification: a bibliography. *SIGSOFT Software Engineering Notes*, 28(6):2–2, 2003. Cité page 34.

[30] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *OSDI*, pages 137–150. USENIX Association, 2004. Cité page 3.

[31] E. W. Dijkstra. In P. Brinch Hansen, editor, *The origin of concurrent programming*, chapter Cooperating sequential processes, pages 65–138. Springer, 2002. Cité page 28.

[32] A. Dragojević, P. Felber, V. Gramoli, and R. Guerraoui. Why STM can be more than a research toy ? *Commun. ACM*, 54:70–77, Apr. 2011. Cité page 5.

[33] U. Drepper. Parallel programming with transactional memory. *Commun. ACM*, 52:38–43, Feb. 2009. Cité page 4.

[34] R. Ennals. Software transactional memory should not be obstruction-free. Technical Report IRC-TR-06-052, Intel Research Cambridge Tech Report, Jan 2006. Cité page 29.

[35] H. González-Vélez and M. Leyton. A survey of algorithmic skeleton frameworks: high-level structured parallel programming enablers. *Software, Practrice & Experience*, 40(12):1135–1160, 2010. Cité page 3.

[36] J. Gray. The transaction concept: Virtues and limitations. In *VLDB*, pages 144–154. VLDB Endowment, 1981. Cité page 28.

[37] D. Grove, O. Tardieu, D. Cunningham, B. Herta, I. Peshansky, and V. Saraswat. A performance model for X10 applications: what's going on under the hood ? In *Proceedings of the 2011 ACM SIGPLAN X10 Workshop*, pages 1–8. ACM, 2011. Cité page 4.

[38] R. L. Halpert, C. J. F. Pickett, and C. Verbrugge. Component-based lock allocation. In *PACT*, pages 353–364. IEEE Computer Society, 2007. Cité page 32.

[39] T. Harris and K. Fraser. Language support for lightweight transactions. *SIGPLAN Not.*, 38:388–402, October 2003. Cité page 29.

[40] T. Harris, S. Marlow, S. Peyton-Jones, and M. Herlihy. Composable memory transactions. In *PPoPP*, pages 48–60. ACM, 2005. Cité page 29.

[41] M. Herlihy, V. Luchangco, and M. Moir. A flexible framework for implementing software transactional memory. In *OOPSLA*, pages 253–262. ACM, 2006. Cité page 30.

[42] M. Herlihy, V. Luchangco, M. Moir, and W. N. Scherer, III. Software transactional memory for dynamic-sized data structures. In *PODC*, pages 92–101. ACM, 2003. Cité page 30.

[43] M. Herlihy and J. E. B. Moss. Transactional memory: architectural support for lock-free data structures. *SIGARCH Comput. Archit. News*, 21(2):289–300, May 1993. Cité page 29.

[44] M. P. Herlihy and J. M. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, July 1990. Cité page 29.

[45] M. Hicks, J. S. Foster, and P. Pratikakis. Lock inference for atomic sections. In *Transact*, Ottawa, Canada, 2006. Cité page 32.

[46] C. Hoare. Towards a theory of parallel programming. In Hoare and Perott, editors, *Operating Systems Techniques*. Academic Press, 1972. Cité page 28.

[47] A. Hobor, A. W. Appel, and F. Z. Nardelli. Oracle Semantics for Concurrent Separation Logic. In S. Drossopoulou, editor, *ESOP*, number 4960 in LNCS, pages 353–367. Springer, 2008. Cité page 34.

[48] W. A. Howard. The formulae-as-types notion of construction. In J. P. Seldin and J. R. Hindley, editors, *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, pages 479–490. Academic Press, 1980. Cité page 9.

[49] S. Jagannathan, J. Vitek, A. Welc, and A. Hosking. A transactional object calculus. *Sci. Comput. Program.*, 57:164–186, August 2005. Cité pages 30 et 32.

[50] G. Klein and T. Nipkow. A machine-checked model for a Java-like language, virtual machine, and compiler. *ACM Trans. Program. Lang. Syst.*, 28(4):619–695, 2006. Cité page 34.

[51] G. Korland, N. Shavit, and P. Felber. Noninvasive concurrency with Java STM. *MultiProg'10 : Programmability Issues for Heterogeneous Multicores*, january 2010. Cité page 31.

[52] E. Koskinen, M. Parkinson, and M. Herlihy. Coarse-grained transactions. In *POPL*, pages 19–30. ACM, 2010. Cité page 31.

[53] J. Larus and C. Kozyrakis. Transactional memory. *Commun. ACM*, 51(7):80–88, 2008. Cité page 4.

[54] X. Leroy. Formal verification of a realistic compiler. *Commun. ACM*, 52(7):107–115, 2009. Cité page 34.

[55] X. Leroy. A formally verified compiler back-end. *Journal of Automated Reasoning*, 43(4):363–446, 2009. Cité pages 23, 24 et 34.

[56] X. Leroy and S. Blazy. Formal verification of a C-like memory model and its uses for verifying program transformations. *Journal of Automated Reasoning*, 41(1):1–31, 2008. Cité page 34.

[57] M. Leyton. *Advanced features for algorithmic skeleton programming*. PhD thesis, Université de Nice Sophia Anti Polis, 2008. Cité page 3.

[58] A. Lochbihler. Verifying a Compiler for Java Threads. In A. D. Gordon, editor, *ESOP*, number 6012 in LNCS, pages 427–447. Springer, 2010. Cité page 34.

[59] M. Martin, C. Blundell, and E. Lewis. Subtleties of transactional memory atomicity semantics. *IEEE Comput. Archit. Lett.*, 5(2):17–17, July 2006. Cité page 43.

[60] J. McCarthy and J. Painter. Correctness of a compiler for arithmetic expressions. In *Proceedings of Symposium in Applied Mathematics, vol. 19, Mathematical Aspects of Computer Science*, pages 33–41. American Mathematical Society, 1967. Cité page 33.

[61] B. McCloskey, F. Zhou, D. Gay, and E. Brewer. Autolocker: synchronization inference for atomic sections. In *POPL*, pages 346–358. ACM, 2006. Cité page 32.

[62] D. McCracken. POSIX threads and the linux kernel. In *Ottawa Linux Symposium*, pages 330–337, 2002. Cité page 28.

[63] Message Passing Interface Forum. MPI: A message-passing interface standard version 3.0, September 2012. `http://www.mpi-forum.org/docs/mpi-3.0`. Cité page 3.

[64] K. F. Moore and D. Grossman. High-level small-step operational semantics for transactions. In *POPL*, pages 51–62. ACM, 2008. Cité pages 33 et 38.

[65] J. E. B. Moss and A. L. Hosking. Nested transactional memory: model and architecture sketches. *Sci. Comput. Program.*, 63:186–201, December 2006. Cité page 30.

[66] N. Hilfinger and Dan Bonachea and Kaushik Datta and David Gay and Susan Graham and Amir Kamil and Ben Liblit and Geoff Pike and Jimmy Su and Katherine Yelick. Titanium Language Reference Manual Version 2.20, August 2006. `http://titanium.cs.berkeley.edu/doc/lang-ref.pdf`. Cité page 4.

[67] H. R. Nielson and F. Nielson. *Semantics with Applications – An Appetizer*. Springer, 2007. Cité page 59.

[68] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*. LNCS 2283. Springer, 2002. Cité page 34.

[69] R. Nishtala, G. Almási, and C. Cascaval. Performance without pain = productivity: data layout and collective communication in UPC. In S. Chatterjee and M. L. Scott, editors, *PPoPP*, pages 99–110. ACM, 2008. Cité page 4.

[70] R. W. Numrich and J. Reid. Co-array Fortran for parallel programming. *SIGPLAN Fortran Forum*, 17(2):1–31, 1998. Cité page 4.

[71] M. Odersky, L. Spoon, and B. Venners. *Programming in Scala*. Artima, second edition, 2010. Cité page 2.

[72] C. H. Papadimitriou. The serializability of concurrent database updates. *Journal of the ACM*, 26(4):631–653, 1979. Cité page 47.

[73] S. Pelagatti. *Structured Development of Parallel Programs*. Taylor & Francis, 1998. Cité page 3.

[74] C. Perfumo, N. Sönmez, S. Stipic, O. Unsal, A. Cristal, T. Harris, and M. Valero. The limits of software transactional memory (STM): dissecting Haskell STM applications on a many-core environment. In *5th Conference on Computing Frontiers (CF'o8)*, pages 67–78. ACM, 2008. Cité page 5.

[75] A. Prokopec, P. Bagwell, T. Rompf, and M. Odersky. A Generic Parallel Collection Framework. In E. Jeannot, R. Namyst, and J. Roman, editors, *Euro-Par*, volume 6853 of *LNCS*, pages 136–147. Springer, 2011. Cité page 3.

[76] F. A. Rabhi and S. Gorlatch, editors. *Patterns and Skeletons for Parallel and Distributed Computing*. Springer, 2003. Cité page 3.

[77] J. Sevcík, V. Vafeiadis, F. Z. Nardelli, S. Jagannathan, and P. Sewell. CompCertTSO: A Verified Compiler for Relaxed-Memory Concurrency. *Journal of the ACM*, 60(3):22, 2013. Cité pages 34 et 35.

[78] J. Sevcik, V. Vafeiadis, F. Zappa Nardelli, S. Jagannathan, and P. Sewell. Relaxed-Memory Concurrency and Verified Compilation. In *POPL*, pages 43–54. ACM, 2011. Cité page 34.

[79] P. Sewell, S. Sarkar, S. Owens, F. Z. Nardelli, and M. O. Myreen. x86-TSO: a rigorous and usable programmer's model for x86 multiprocessors. *Commun. ACM*, 53(7):89–97, 2010. Cité page 34.

[80] N. Shavit and D. Touitou. Software transactional memory. In *PODC*, pages 204–213. ACM, 1995. Cité pages 4 et 29.

[81] M. Snir and W. Gropp. *MPI the Complete Reference*. MIT Press, 1998. Cité page 3.

[82] The Coq Development Team. The Coq Proof Assistant. `http://coq.inria.fr`. Cité pages 9 et 50.

[83] UPC Consortium. UPC language specifications version 1.2, May 2005. `http://www.gwu.edu/~upc/publications/LBNL-59208.pdf`. Cité page 4.

[84] Vijay Saraswat and Bard Bloom and Igor Peshansky and Olivier Tardieu and David Grove. X10 language specification version 2.3, February 2013. `http://x10.sourceforge.net/documentation/languagespec/x10-231.pdf`. Cité page 4.

[85] K. A. Yelick, L. Semenzato, G. Pike, C. Miyamoto, B. Liblit, A. Krishnamurthy, P. N. Hilfinger, S. L. Graham, D. Gay, P. Colella, and A. Aiken. Titanium: A High-performance Java Dialect. *Concurrency - Practice and Experience*, 10(11-13):825–836, 1998. Cité page 4.

Ce document a ecrit a l'aide de texte GNU Emacs et du logiciel de composition typographique LaTeX 2ε.

Thomas Pinsard

Sections atomiques emboîtées avec échappement de processus légers : sémantiques et compilation

Résumé :

La mémoire transactionnelle est un mécanisme de plus en plus populaire pour la programmation parallèle et concurrente. Dans la plupart des implantations, l'emboîtement de transactions n'est pas possible ce qui pénalise la modularité. Plutôt que les transactions, qui sont un choix possible d'implantation, nous considérons directement la notion de section atomique. Dans un objectif d'améliorer la modularité et l'expressivité, nous considérons un langage impératif simple étendu avec des instructions de parallélisme avec lancement et attente de processus légers et une instruction de section atomique à portée syntaxique, depuis laquelle des processus légers peuvent s'échapper.
Dans ce contexte notre première contribution est la définition précise de l'atomicité et de la bonne synchronisation. Nous prouvons que pour des traces bien formées, la dernière impliquent la forme forte de la première. Ceci est fait sur des traces d'exécution abstraites dans le sens où nous ne définissons par précisément la syntaxe et la sémantique opérationnelle d'un langage de programmation. Cette première partie de notre travail peut être considérée comme une spécification pour un tel langage. Nous avons utilisé l'assistant de preuve Coq pour modéliser et prouver nos résultats. Notre deuxième contribution est la définition formelle du langage $Atomic\ Fork\ Join$ (AFJ). Nous montrons que les traces de sa sémantique opérationnelle vérifie effectivement les conditions de bonne formation définies précédemment. La troisième contribution est la compilation de programmes AFJ en programmes $Lock\ Unlock\ Fork\ Join$ (LUFJ) un langage avec processus léger et verrous mais sans sections atomiques. Nous étudions la correction de la compilation de AFJ vers LUFJ.

Mots clés : sémantique formelle, parallélisme, processus léger, section atomique, verrou, bonne synchronisation, trace de programme, atomicité, assistant de preuve, sémantique opérationnelle, compilation, préservation de la sémantique

Nested Atomic Sections with Thread Escape : Semantics and Compilation

Abstract :

Transactions are becoming a popular mechanism for parallel and concurrent programming. In most implementations the nesting of transactions is not supported which hinders modularity. Rather than transactions, which are an implementation choice, we consider directly the notion of atomic section. For the sake of modularity with we consider a simple imperative language with fork/join parallelism and lexically scoped nested atomic sections from which threads can escape.
In this context, our first contribution is the precise definition of atomicity, well-synchronisation and the proof that the latter implies the strong form of the former. This is done on execution traces without being specific to a language syntax and operational semantics. This first part of our work could be considered as a specification for the design and implementation of such a parallel language. A formalisation of our results in the Coq proof assistant is also available. Our second contribution is a formal definition of the Atomic Fork Join (AFJ) language and its operational semantics. We show that it indeed satisfies the conditions previously defined. The third contribution of our work is a compilation procedure of AFJ programs to programs another language with threads and locks but without atomic sections, named Lock Unlock Fork Join (LUFJ). We study the correctness of the compilation from AFJ to LUFJ.

Keywords : formal semantics, parallelism, thread, atomic section, lock, well-synchronisation, program trace, atomicity, proof assistant, operational semantics, compilation, semantic preservation

Laboratoire d'Informatique Fondamentale d'Orléans
(LIFO)
Université d'Orléans - Faculté des Sciences
Bâtiment IIIA
Rue Léonard de Vinci
B.P. 6759
F-45067 ORLEANS Cedex 2, France